Programming by multiset transformation

Jean-Pierre Banâtre, Daniel Le Métayer

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1)39 63 55 11

Rapports de Recherche

N° 1205

Programme 3
Réseaux et Systèmes Répartis

PROGRAMMING BY MULTISET TRANSFORMATION

Jean-Pierre BANATRE
Daniel LE METAYER

Avril 1990

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CÉDEX
FRANCE
Téléphone : 99 36 20 00
Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

Programming by multiset transformation

Programmation par transformation de multi-ensembles

Jean-Pierre Banâtre
Daniel Le Métayer

Publication Interne n°522 - mars 1990 - 26 pages

Abstract: *We present a new formalism called GAMMA in which programs are described in terms of multiset transformations. A distinguishing property of GAMMA is the possibility of expressing algorithms in a very abstract way, without any artificial sequentiality. The expressive power of the formalism is illustrated through a series of examples chosen from a wide range of domains (string processing problems, graph problems, geometric problems ...).*

Résumé: *Nous présentons un nouveau formalisme, appelé GAMMA, dans lequel les programmes sont décrits comme des transformateurs de multi-ensembles. La caractéristique majeure de GAMMA est de pouvoir exprimer des algorithmes sous une forme très abstraite, sans séquentialité artificielle. La puissance d'expression du formalisme est illustrée par une série d'exemples choisis dans divers domaines d'applications (traitement de textes, problèmes de graphes, problèmes de géométrie, ...).*

Ce travail a été effectué à l'IRISA, dans le cadre de l'équipe *Langages et Systèmes Parallèles.*

Adresse des auteurs: IRISA, Campus de Beaulieu, 35042 RENNES CEDEX, FRANCE

CENTRE NATIONAL DE LA RECHERCHE SCIENTIFIQUE
(L. A. 227)
UNIVERSITÉ DE RENNES 1 I. N. S. A. DE RENNES

INSTITUT NATIONAL DE RECHERCHE
EN INFORMATIQUE ET EN AUTOMATIQUE
(LABORATOIRE DE RENNES)

It has been argued for a long time that parallelism in a programming language makes the programming task far more difficult. This statement is undoubtedly true in the context of imperative languages because the programmer has to mentally manage several threads of control. Let us remark however that even sequential imperative programming entails an operational reasoning which is difficult to master. This suggests that the origin of the problem lies in the imperative paradigm rather than in parallelism itself. We go even further: we believe that parallelism is a powerful structuring facility that could profitably be exploited in program construction. We should make a distinction here between *logical parallelism* and *physical parallelism*. Physical parallelism is related to the implementation: it corresponds to the distribution of tasks on several processors. By logical parallelism, we mean the possibility of describing a program as a composition of several independent tasks. Of course, a particular implementation can turn logical parallelism into physical parallelism but these two notions have very different natures: the former is a program structuring tool whereas the latter is an implementation technique. Unfortunately, the term *parallelism* is often used without distinction for both of these concepts, which makes any discussion about parallelism very confusing. In this paper we use the term *parallelism* for *logical parallelism*; we are not concerned with implementation issues.

In fact the confusion between these two kinds of parallelism is part of the heritage of several decades of imperative programming. Another aspect of this lack of discrimination is the pervasive use of the sequencing operator ";" in imperative programming. This operator has been introduced because imperative programming languages were supposed to be "abstract" models of traditional von Neumann machines. As a consequence, most ";" in imperative programs reflect the need to model sequential machines, whereas few are really imposed by the logic of the program.

For example let us consider the following program fragment:

ma := max_array(a);
mb := max_array(b);
m := max(ma,mb)

where *max_array* is supposed to be a user-defined function yielding the maximum element of an array. The first occurrence of ";" is not related to the logic of the program (the first two statements could be exchanged without altering the meaning of the program); it is imposed by the (supposed) underlying architecture. The second occurrence of ";" however expresses a logical dependency between the first two statements and the last one (exchanging them would alter the meaning of the program).

This distinction is related to the more general issue of the separation of correctness and efficiency concerns: it is now admitted that correctness should be the primary concern in program development; it is only in a second stage that provably correct solutions should be used to derive more efficient versions. To this respect let us quote [6]: *"The basic problem in programming is managing complexity. We cannot address that problem as long as we lump together concerns about the core problem to be solved, the language in which the program is to be written, and the*

hardware on which the program is to execute. Program development should begin by focusing attention on the problem to be solved and postponing considerations of architecture and language constructs". In order to achieve this goal, one should be able to build in the first place an abstract version of the program in a high level language. In particular, these abstract programs should be free of artificial sequentiality. Unfortunately, to our knowledge, there is no available formalism allowing such an abstract description of programs. Let us take a simple example to illustrate this point. The problem is to find the maximum element of a set. In an imperative language the set can be represented as an array *a[1:n]* and a possible program is:

$maxset_1$: $m := a[1];$
$i := 1;$
$* [i < n \rightarrow$
$i := i+1;$
$m := max(m, a[i])]$

While the condition $i < n$ holds, index i is incremented and a new value of m is computed.
In a functional language, the set would be represented as a list and the program would be:

$maxset_2(l)$ = ***if*** $tail(l) = nil$ ***then*** $head(l)$ ***else*** $max\ (head(l), maxset_2(tail(l)))$

In both cases the program imposes a strict ordering between the comparisons of the elements: the first element is compared with the second, then their maximum is compared with the third, and so forth... In both formalisms one could imagine a less constraining solution involving implicit parallelism. For example, in the functional language a divide-and-conquer version of the above program would be:

$maxset_3(l)$ = ***if*** $tail(l) = nil$ ***then*** $head(l)$ ***else***
let (l_1, l_2) = $split(l)$ ***in***
$max(maxset_3(l_1), maxset_3(l_2))$

where $split\ (l_1, ..., l_n) = ((l_1, ..., l_{n/2}), (l_{n/2+1}, ..., l_n))$. Here again a (non-strict) ordering is imposed on the comparisons: for example the first and last elements of the list will not be compared (except in the case where they are the maxima of their respective sublists).

In fact the maximum of a set can be computed by performing the comparisons of the elements in any order. So we would like an abstract algorithm of the form:

while *there are at least two elements in the set*
select *two elements of the set,* ***compare*** *them and* ***remove*** *the smaller one*

This is almost a GAMMA program. In GAMMA such a statement can be written as follows:

$maxset_4\ (s) = \Gamma((R,A))\ (s)$ **where**

$R(x,y) = x \leq y$

$A(x,y) = \{y\}$

Function R specifies a property to be satisfied by the selected elements; these elements are replaced in the set by the result of the application of function A. Nothing is said in this definition about the order of evaluation of the comparisons; if several disjoint pairs of elements satisfy property R, the comparisons and replacements can even be performed in parallel. An intuitive way of describing the meaning of a GAMMA program is the metaphor of the chemical reaction: the set can be seen as a chemical solution, function R (called the reaction condition) is a property to be satisfied by reacting elements and A (the action) describes the product of the reaction. The computation terminates when a stable state is reached, that is to say when no elements of the set satisfy the reaction condition.

Let us now give a more formal presentation of GAMMA. The basic data structure in GAMMA (General Abstract Model for Multiset mAnipulation) is the *multiset*, which is the same as a set except that it may contain multiple occurrences of the same element [16]; the multiset is sometimes referred to as a *bag*. The benefit of using multisets is the possibility of describing compound data without any form of constraint or hierarchy between its components. This is not the case for recursively defined data structures such as lists which impose an ordering on the examination of elements (function $maxset_2$ above is an illustration of this constraint). The control structure associated with multisets is the Γ operator; as we have seen on the above example, Γ reflects the absence of hierarchy in the data structure and entails some kind of chaotic model of execution. Its formal definition can be stated as follows:

$\Gamma((R_1,A_1),...,(R_m,A_m))\ (M) =$

if $\quad \forall i \in [1,m],\ \forall x_1,...,x_n \in M,\ \neg R_i\ (x_1,...,x_n)$

then $\quad M$

else $\quad$ **let** $x_1,...,x_n \in M$, **let** $i \in [1,m]$ **such that** $R_i\ (x_1,...,x_n)$ **in**

$\Gamma((R_1,A_1),...,(R_m,A_m))\ ((M - \{x_1,...,x_n\}) + A_i\ (x_1,...,x_n))$

The notation $\{..\}$ is used to represent multisets. There is no ambiguity here since we never use simple sets. The basic operations on multisets are the following:

- *union*: the number of occurrences of an element in $M_1 + M_2$ is the sum of its numbers of occurrences in M_1 and M_2.
- *difference*: the number of occurrences of an element in $M_1 - M_2$ is the difference between its numbers of occurrences in M_1 and M_2 (if this difference is greater than or equal to zero, otherwise it is zero).
- *product*: $M \times N$ is the cartesian product of M and N.
- *maximum*: the number of occurrences of an element in $M \cup N$ is the maximum of its numbers of occurrences in M and N.
- *minimum*: the number of occurrences of an element in $M \cap N$ is the minimum of its numbers of occurrences in

M and N.

- *cardinal*: *card(M)* yields the sum of the numbers of occurrences of all the elements of the multiset M.

We use $\exists x_1,...,x_n \in M$ as a shorthand notation for $\exists \{x_1,...,x_n\} \subseteq M$, which means that $x_1,...,x_n$ are different elements of the multiset (even if some of them may possibly possess the same value); this is in contrast to $\exists x_1 \in M,..., \exists x_n \in M$, where $x_1,...,x_n$ are not necessarily different elements.

(R_i,A_i) are pairs of closed functions (functions whose definition does not involve global variables) specifying reactions. The effect of a reaction (R_i,A_i) on a multiset M is to replace in M a subset of elements $\{x_1,...,x_n\}$ such that $R_i(x_1,...,x_n)$ is true by the elements of $A_i(x_1,...,x_n)$. If no elements of M satisfy any reaction condition ($\forall i \in [1,m], \forall x_1,...,x_n \in M, \neg R_i(x_1,...,x_n)$) then the result is M; otherwise the result is obtained by carrying out one reaction $((M - \{x_1,...,x_n\}) + A_i(x_1,...,x_n))$ and repeating the same process. The above definition implies that if one or several reaction conditions hold for several subsets at the same time, the choice which is made among them is not deterministic. The importance of the *locality property* of GAMMA cannot be overemphasized: if the reaction condition holds for several disjoint subsets, the reactions can be carried out independently (and simultaneously). This property is the basic reason why GAMMA programs do generally exhibit a lot of potential parallelism.

GAMMA programs can be composed using the traditional functional notation. Furthermore, functions R_i and A_i can be defined by pattern matching on the form of their arguments; pattern matching can also be used to extract elements from a multiset (for example m ***where*** $\{m\} = \Gamma((R,A))(G)$ is the extraction of the unique element of the result multiset). It should be noticed that the arguments of R_i and A_i must be of the same form since A_i is applied to arguments satisfying R_i. The notation used above to describe GAMMA programs may seem slightly awkward since it requires two separate definitions for R_i and A_i which entails the duplication of the text of their formal arguments. An alternative syntax could be:

replace $(x_1,...,x_n)$ ***such that*** *"text of the condition"*
by *"text of the action"*

Nevertheless we shall keep the original syntax here because it emphasizes the functional nature of reaction conditions and actions.

We discuss in next section the programming style entailed by the GAMMA formalism. Then we illustrate the expressive power of the language with a variety of problems: numerical problems, sorting problems, string processing problems, graph problems, geometric problems, and process synchronization problems. The conclusion contains a comparison with related work and a discussion of the role of a high level language like GAMMA in the program construction process.

THE GAMMA STYLE OF PROGRAMMING

In this section we illustrate three programming styles through a simple example: the imperative style, the functional style and the GAMMA style. Our objective is to emphasize the distinguishing properties of GAMMA and to exhibit the basic programming methodology in GAMMA.

Prime number generation

The goal is to produce the prime numbers less than a given N.

Imperative solutions

These solutions are based on standard sieve techniques, which rely on the fact that the $(i+1)^{th}$ prime number (with $i \geq 1$) is the smallest integer exceeding the i^{th} prime that is not divisible by the first i primes. Integers greater than $\sqrt{N}$ which have not been eliminated by divisions by integers less than $\sqrt{N}$ are prime. We present two solutions in a traditional imperative language with guarded commands: a sequential program and a program with explicit parallelism.

Sequential solution

Initially array t contains 2 and all odd integers less than N, and eventually the first k_max elements of t contain the primes less than N. $\lceil X \rceil$ represents the smallest integer greater than X.

```
begin
   var max: integer init ⌈(N+1)/2⌉;
   t: array[1: max] of integer;
   var k_max: integer init max;
   var n: integer init 3;
   var i: integer init 2;
   var current_odd: integer init 2;
    var p, k, m: integer;
   t[1] := 2;
   *[ i ≤ max → t[i] := n ; i:= i+1; n:= n+2];
   *[ t[current_odd] < √N →
         i := current_odd+1; k := i;
         p:=t[current_odd];
         *[ i≤ k_max →
               m := t[i];
               [multiple(m,p) → skip □ not(multiple(m,p)) → t[k] := m; k:= k+1];
               i:= i+1
         ];
         k_max := k;
         current_odd := current_odd+1
   ]
end
```

To simplify this solution, we have used function *multiple(x,y)* which returns *True* if x is a multiple of y and *False* otherwise. The invariant maintained by this program is: *(∀i ∈ [1: current_odd], t[i] ∈ {primes less than N})*. The program is made of two embedded loops; iteration eliminates numbers between *t[current_odd + 1]* and *t[k_max]* which are multiple of the prime *t[current_odd]*; *t[current_odd]* corresponds to the last detected prime, i is the index of the element of the array currently being processed and k is the index of the last element in the current array which is not a multiple of *t[current_odd]*.

This sequential program is rather hard to understand. It involves several state variables used to manage the computation of primes and the necessary updatings of t. Shorter imperative programs can be proposed but they are not significantly simpler and we have chosen to present this one because it is closer to the functional and GAMMA solutions.

Parallel solution

This solution is expressed in CSP [15] and it is inspired by a program given in [15]. It is defined in terms of an array of processes, *sieve*, in which each process first inputs a prime p from its predecessor, prints it, and then transmits to its successor all the numbers received from its predecessor that are not multiples of p. There is no dynamic creation of processes in CSP; so it is necessary to determine statically the exact number of processes necessary to carry out the computation. In fact, $\lceil\sqrt{N}\rceil + 2$ processes are necessary: one process computes the even integers, $\lceil\sqrt{N}\rceil$ processes compute the primes less than $\lceil\sqrt{N}\rceil$ and one process collects integers greater than $\lceil\sqrt{N}\rceil$ and not eliminated. An extra process named *print* is supposed to be available to gather the primes and to print them.

```
begin
   sieve[0] ::
      print ! 2; n: integer; n:= 3;
      *[ n ≤ N → sieve[1] ! n; n:= n+2]
   ||
   sieve(i:1.. ⌈√N⌉) ::
      p: integer;
      sieve(i-1) ? p;
      print ! p;
      *[ m: integer; sieve(i-1) ? m →
          [multiple(m,p) → skip□ not(multiple(m,p)) → sieve(i+1)!m]
       ]
   ||
   sieve[ ⌈√N⌉ + 1] ::
      *[n: integer; sieve[ ⌈√N⌉] ? n → print ! n]
end
```

Functional solution

This recursive solution uses two auxiliary functions: *int_between(m, n)* which produces the list of integers *(m..n)* and *filter(p, x)* which eliminates all multiples of *p* from list *x*.

primes (N) = prime_numbers (int_between (2, N))

filter (p, y) = if multiple (head (y), p) then filter (p, tail (y))
else cons (head(y), filter(p, tail(y)))

int_between (m, n) = if m > n then nil
else cons (m, int_between (m+1, n))

prime_numbers (l) = if null(l) then nil
else cons (head(l), prime_numbers(filter(head(l), tail(l)))

GAMMA solution

In GAMMA the solution consists in removing multiple elements from the multiset *{2,...,N}*. The result multiset contains exactly the prime numbers less than *N*: a prime number cannot be eliminated from the initial multiset and eliminated elements are obviously not prime numbers.

prime_numbers(N) = Γ((R, A)) ({2, ..., N}) **where**
R(x, y) = multiple(x, y)
A(x, y) = {y}

Compared with the previous solutions, the GAMMA program is far more concise and easier to understand. The reason is that many computational details are left unspecified in GAMMA. Both imperative solutions are more intricate because they give a precise description of execution order (expressed through the ";" operator in the sequential program and through explicit process communication in the parallel program) and memory management (through the use of a collection of variables). The functional solution is also lower level because list construction and decomposition have to be made explicit. In fact, one could say that the GAMMA program captures in a natural way the basic strategy applied in the imperative and functional solutions, namely to eliminate multiple elements from the set *{2,...,N}*.

As a consequence of the absence of commitment to a particular execution order, the GAMMA program can be implemented naturally in a parallel way [2]. This is also the case for the parallel imperative program but the major difference is that the programmer is responsible for parallelism management and he must have a precise knowledge of the underlying architecture to write a program suitable for parallel execution. This is a case where conflicts may arise between logical parallelism used as a programming technique and the physical parallelism necessary to exploit a particular architecture. The functional solution can also be implemented in a parallel way: functions *int_between*, *filter* and *prime_numbers* can be executed in a pipeline but the parallelism is far more constrained than in the GAMMA program where all comparisons between different elements can potentially be performed simultaneously. It should be clear that we do not claim here that a realistic parallel implementation of GAMMA is straightforward. In

fact any reasonable implementation of GAMMA must rely on sophisticated derivation techniques. What we claim is that our high level of abstraction eliminates unfortunate sequential biases.

Program construction in GAMMA

The GAMMA model of computing puts forward a quite unusual approach to program design. A program is no longer a sequence of instructions modifying a state, or a function applied to its arguments but rather a multiset transformer operating on all the data at once. The development of GAMMA programs entails a choice in data representation and a choice in the type of transformation applied to this data.

Data decomposition

The unique data structure provided in GAMMA is the multiset. Elements of multisets may be basic or composed values (even multisets) but there is no recursive data structure definition in GAMMA. So the first task to perform when designing a program in GAMMA is to find a representation of data as a multiset of items. If the program has to operate on basic values such as integers, a suitable decomposition of these values has to be found. For example the *prime_numbers* program above decomposes its argument N into a multiset $\{2,...,N\}$. Next section contains another decomposition of integers in terms of prime factors. Complex data such as sequences, trees or graphs can be represented as multisets in a straightforward way: for example, the components of a multiset representing a tree are the nodes and the leaves of the tree associated with parenthood information, the components of a sequence multiset are pairs *(index,value)* ... It is a distinguishing property of GAMMA to view all data structures as flat multisets: all the components of a data structure are directly accessible, independently of their position in the structure. If the position in the structure is relevant to the reaction, it has to be expressed via the reaction condition ($R(x,y)$ = *leftson(x,y)* for example, in a tree manipulating program). We can say that GAMMA has a *topological view of data types*; this contrasts with the traditional *recursive view of data types* where a walk through the data is necessary to access a particular component. This property has deep effects on the GAMMA programming style.

Relaxation

Relaxation is a method used in mathematics to solve systems of equations by iteration; first an estimate vector solution is produced (guessed), and the errors in the initial estimation are decreased and relaxed as calculation continues. The same method may be applied for solving many problems with the GAMMA paradigm. The initial multiset represents a (possibly rough) estimate of the solution and a series of actions refine this estimate until a proper solution is found. In the prime number generator example, the initial multiset $\{2, ..., N\}$ can be seen as a rough estimate of the set of prime numbers less than N, and the computation refines the result by eliminating the anomalies (non primes).

Data expansion and reduction

Programming by relaxation is suitable when the basic structure of the result is known. Let us now take a new example to illustrate two other programming techniques: data expansion and reduction. The following GAMMA program computes *Fibonacci(n)* where *Fibonacci(n)* = ***if*** $n \leq 1$ ***then*** *1* ***else*** *Fibonacci(n-1)+Fibonacci(n-2)*:

$fib(n) = m$ **where**

$\{m\} = sigma(gen(\{n\}))$

$gen(N) = \Gamma((R_1, A_1), (R_2, A_2))(N)$ **where**

$R_1(n) = n > 1$

$A_1(n) = \{n - 1, n - 2\}$

$R_2(0) = true$

$A_2(0) = \{1\}$

$sigma(M) = \Gamma(R, A)(M)$ **where**

$R(x, y) = true$

$A(x, y) = \{x + y\}$

The initial number n is decomposed into a number of ones which are then summed up to produce the expected result. By *data expansion*, we mean the decomposition of values into a collection of items. Computation stops when the multiset is a collection of indivisible elements. Data expansion involves unary reaction conditions and actions. In the *fib* example above, *gen* performs a data expansion, indivisible elements are ones. *Reduction* corresponds to the case where a multiset of items is reduced to a singleton by successive applications of the action. In the Fibonacci example, the *sigma* operator proceeds by reduction: a multiset $\{1, ..., 1\}$ is transformed into a singleton multiset by a series of sums.

Data expansion and data reduction are dual programming techniques: the former decomposes values into collections of simpler components and the latter gathers individual elements to build more complex values. A comparison of the GAMMA program with the original functional version of Fibonacci shows that data expansion corresponds to the recursive function calls and reduction to function returns. On the other hand, relaxation does not transform the structure of the multiset but proceeds by successive refinements. The three programming techniques described here are put into practice in the rest of the paper to solve a wide range of problems.

NUMERICAL PROBLEMS

Numerical problems are often used to test the expressiveness of linguistic constructs. We have chosen two examples here: the classical factorial problem and the prime factorization of an integer.

Factorial

The following GAMMA program computes $n!$:

$fact(n) = \Gamma((R,A))(\{1, ..., n\})$ **where**

$R(x, y) = true$

$A(x, y) = \{x*y\}$

This very simple program is an illustration of the reduction technique. It should be noticed that no constraint is put on the order in which multiplications are performed. No imperative or functional solution exhibits such a freedom in

the execution order.

Prime factorization of an integer

A fundamental theorem of arithmetic states that every positive integer n can be written as a product of primes, and that this decomposition is unique. This fact gives a one-to-one correspondance between positive integers and multisets of prime numbers; for example if $n = 2^2 * 3^3 * 11$, the corresponding multiset is $\{2, 2, 3, 3, 3, 11\}$.

Given a positive integer n, the set of prime numbers less than or equal to n, can be obtained using the program *primes* presented earlier. The notation $P \otimes \{(n, 0)\}$ represents the set of triples $(p_i,n,0)$, where $p_i \in P$.

$factorization(n) = P_2\ (P_1(\ prime\_numbers(n) \otimes \{(n, 0)\}))$ **where**

$P_1(M) = \Gamma((R_1, A_1))(M)$

$P_2(M) = \Gamma((R_2, A_2))\ (M)$

$R_1((n_1, n_2, k)) = multiple(n_2, n_1)$

$A_1((n_1, n_2, k)) = \{(n_1, n_2/n_1, k+1)\}$

$R_2((n_1, n_2, k)) = true$

$A_2((n_1, n_2, k)) = \{n_1,...,n_1\}$ *(multiset with k occurrences of* n_1*)*

P_1 proceeds by relaxation to evaluate the coefficient associated with each prime number and P_2 removes unnecessary information from the triples. The prime factorization of the greatest common divisor, the least common multiplier and the product of two numbers can be computed very easily from their prime factorization:

$gcd\ (M,N) = M \cap N$

$lcm(M,N) = M \cup N$

$product(M,N) = M + N$

SORTING PROBLEMS

We illustrate the relaxation principle through three instances of the general problem of sorting a collection of values according to a particular criterion.

Multiset partitioning

Given two multisets of integers S and T, the problem consists in partitioning $S + T$ into two multisets S' and T' such that $S + T = S' + T'$, $card(S) = card(S')$ and $card(T) = card(T')$, and every element of S' is smaller than or equal to every element of T'. This is a generalization to multisets of the traditional set partitioning problem. The solution in GAMMA consists in gathering all the elements into a multiset $\{(x,inS) / x \in S\} + \{(x,inT) / x \in T\}$ and exchanging values (x,inS) and (y,inT) such that $x > y$ until the solution is reached (inS and inT are tags representing the origin of the value).

$sp(S,T) = \Gamma((R,A))\ (S \times \{inS\} + T \times \{inT\})$ **where**

$R((x,inS),(y,inT)) = x > y$

$A((x,inS),(y,inT)) = \{(x,inT),(y,inS)\}$

Traditional solutions to this problem proceed sequentially by selecting the greatest element of S and the smallest element of T and exchanging them until the former is smaller than or equal to the latter. In contrast, our solution performs exchanges in a chaotic way and the tag associated with a value may switch several times.

Dutch national flag

This variation of the partitioning problem has been proposed by Dijkstra. The goal is to sort an array of elements designated *red*, *white* or *blue* so that all the *red* elements appear before the *white*, which in turn appear before all the *blue* elements [10]. We assume that the initial multiset contains at least one element of each colour. In GAMMA, the array is represented by a multiset of pairs *(index,colour)* and the program proceeds again by exchanging ill-sorted elements until the solution is reached.

$dnf(Array) = \Gamma((R,A))\ (Array)$ **where**

$R((i,red),(j,white)) = i > j$

$A((i,red),(j,white)) = \{(i,white),(j,red)\}$

$R((i,white),(j,blue)) = i > j$

$A((i,white),(j,blue)) = \{(i,blue),(j,white)\}$

Sorting

We consider now the general sorting problem: the goal is to organize the elements of an array in increasing order. We use again a multiset of pairs *(index,value)* and the program exchanges ill-ordered values until all values are well-ordered.

$sort(Array) = \Gamma((R,A))\ (Array)$ **where**

$R((i,v),(j,w)) = (i > j)$ **and** $(v < w)$

$A((i,v),(j,w)) = \{(i,w),(j,v)\}$

STRING PROCESSING PROBLEMS

A string can be seen as a linear sequence of characters. Strings are central in many applications of computer science such as word processing systems for example. We present here three well-known string processing examples: the telegram problem, the longest upsequence problem and the majority element problem.

The telegram problem

The telegram analysis problem can be stated as follows [10]: it is required to process a stream of telegrams, each terminated by a string ZZZZ. Words are separated by one or more spaces. The resulting telegram must have a single

space between words and no leading or trailing spaces.

Data items are represented as triples (v, i, s), where v is the value of the data item (a character or a space), i is the index of this item and s is the shift to be used for accessing the next significant data item in the sequence. Initially, s is equal to 1 for all triples. The first character has index 1 and a string ZZZZ is added in the front of the sequence to avoid special treatment for the first telegram.

$$telegram\_analysis\ (M) = \Gamma((R_1, A_1), (R_2, A_2), (R_3, A_3), (R_4, A_4))\ (M + \{(Z,-3,1),(Z,-2,1),(Z,-1,1),(Z,0,1)\})$$

where

$$R_1((space, i_1, s_1), (space, i_2, s_2)) = (i_2 = i_1 + s_1)$$

$$A_1((space, i_1, s_1), (space, i_2, s_2)) = \{(space, i_1, s_1 + s_2\ )\}$$

$$R_2((v_1, i_1, s_1), (v_2, i_2, s_2)) = (i_2 = i_1 + s_1 \textbf{ and } s_1 > 1)$$

$$A_2((v_1, i_1, s_1), (v_2, i_2, s_2)) = \{(v_1, i_1, 1), (v_2, i_1 + 1, s_1 + s_2 - 1)\}$$

$$R_3((Z,i,1),(Z,i+1,1),(Z,i+2,1),(Z,i+3,1),(space,i+4,k)) = True$$

$$A_3((Z,i,1),(Z,i+1,1),(Z,i+2,1),(Z,i+3,1),(space,i+4,k)) =$$
$$\{(Z,i,1),(Z,i+1,1),(Z,i+2,1),(Z,i+3,k+1)\}$$

$$R_4((space,i-1,1),(Z,i,1),(Z,i+1,1),(Z,i+2,1),(Z,i+3,k)) = True$$

$$A_4((space,i-1,1),(Z,i,1),(Z,i+1,1),(Z,i+2,1),(Z,i+3,k)) =$$
$$\{(Z,i-1,1),(Z,i,1),(Z,i+1,1),(Z,i+2,k+1)\}$$

Each reaction corresponds to a particular requirement of the specification: (R_1, A_1) and (R_2, A_2) are used for space elimination ((R_1,A_1) performs the elimination at the price of increasing shifts and (R_2, A_2) compacts the telegram) and (R_3, A_3), (R_4, A_4) for elimination of the first and last spaces. According to our classification, this program performs a relaxation (the initial multiset being a first approximation of the solution).

This problem which is presented in [10] as inherently sequential, is given here a solution with a high potential for parallelism: elimination of extra spaces and compaction may be carried out concurrently and can lead to a fair amount of parallelism.

The longest upsequence problem

A subsequence is obtained from a sequence by deleting some (non necessarily adjacent) values. A sequence is called an upsequence if its values are in non-decreasing order. The problem is to compute the length of the longest upsequence of a sequence.

A sequence is represented as a set of triples (n, x, l_n), where x is the value at index n and l_n is the length of the longest known upsequence ending at index n. We first give a GAMMA program which computes all the triples (n, x, l_n); M_0 is a multiset of pairs representing the initial sequence:

$$lup(M_0) = \Gamma(R, A)\ (\{(k, x_k, 1) \mid (k, x_k) \in M_0\}) \quad \textbf{where}$$

$$R((n, x_n, l_n), (k, x_k, l_k)) = (n < k \textbf{ and } x_n \leq x_k \textbf{ and } l_k < l_n + 1)$$

$$A((n, x_n, l_n), (k, x_k, l_k)) = \{(n, x_n, l_n), (k, x_k, l_n + 1)\}$$

A second program *max* is required to find the maximum of the lengths l_i.

$$max(M) = \Gamma(R_{max}, A_{max})\,(M) \quad \textbf{where}$$
$$R_{max}\,((n, x_n, l_n), (k, x_k, l_k)) = (l_k \leq l_n)$$
$$A_{max}\,((n, x_n, l_n), (k, x_k, l_k)) = \{(n, x_n, l_n)\}$$

The GAMMA program evaluating the longest upsequence is the following:

$$lus(M) = l_k \textbf{ where } \{(k, x_k, l_k)\} = max(lup(M_0))$$

Program *lup* performs a relaxation and *max* is a reduction.

The majority element problem

The majority element of a multiset M is an element occurring more than $card(M)/2$ times in the multiset. We propose first a solution to the problem of finding the majority element, assuming that such an element exists:

$$maj\_elem(M) = \Gamma(R, A)(M) \quad \textbf{where}$$
$$R(x, y) = (x \neq y)$$
$$A(x, y) = \{\}$$

This solution can be seen as an abstract (and parallel) version of the "hands-in-the-pocket" presentation given in [14]. Let us now discharge the assumption of the existence of a majority element. We require a program yielding the majority element if it exists and $\perp$ otherwise. Elements of the original multiset are represented by pairs (v, n), where v is the value and n is the number of occurrences of v represented by the pair; initially $n = 1$ for all values.

$$maj\_elem(M) = P_1(P_2(M \times \{1\}), card(M)) \quad \textbf{where}$$
$$P_2(M) = \Gamma((R_2, A_2))\,(\Gamma((R_1, A_1))(M)) \quad \textbf{where}$$
$$R_1((v_1, n_1), (v_2, n_2)) = (v_1 = v_2)$$
$$A_1((v_1, n_1), (v_2, n_2)) = \{(v_1, n_1 + n_2)\}$$
$$R_2((v_1, n_1), (v_2, n_2)) = (n_1 \geq n_2)$$
$$A_2((v_1, n_1), (v_2, n_2)) = \{(v_1, n_1)\}$$
$$P_1(M, c) = \textbf{if } n > c/2 \textbf{ then } v \textbf{ else } \perp$$
$$\textbf{where } \{(v, n)\} = M$$

This program is an elaborated version of the previous one. The numbers of occurrences are necessary to decide whether the resulting value is a majority element. Reaction (R_1, A_1) performs a relaxation to evaluate the number of occurrences of every value in the multiset and (R_2, A_2) is a reduction yielding an element whose number of occurrences is maximum. This is yet another problem usually presented as inherently sequential which has a nice

GAMMA solution with a high potential for concurrency.

GRAPH PROBLEMS

Many problems are naturally formulated in terms of graphs. We show in this section how three fundamental graph problems can be dealt with in GAMMA: the connectivity problem, the shortest-path problem and the minimum spanning tree problem.

Connectivity

An undirected graph is a collection of *vertices* (or *nodes*) and *edges*. Vertices are simple objects and edges are connections between two vertices. A *path* from vertex x to y is a list of vertices in which successive vertices are connected by edges in the graph. A graph is *connected* if there is a path from every node to every other node. The problem we consider here is to decide whether a graph is connected or not. The idea of the GAMMA program is to build bigger and bigger aggregates of connected nodes: the graph is connected if and only if all the nodes can ultimately be gathered into one aggregate. This program is a typical example of application of the reduction strategy.

Graphs are represented as multisets of vertices and edges: a vertex x is denoted by the singleton $\{x\}$ and an edge connecting x to y is represented by a pair (x,y). We use the boolean function *vertices* to test whether an element of the multiset is a set of vertices. The GAMMA program is the following:

$connected\ (G) = singleton\ (\Gamma((R_1,A_1),(R_2,A_2))\ (G))$ **where**

$R_1(v,w,(m,n)) = vertices(v)$ **and** $vertices(w)$ **and** $m \in v$ **and** $n \in w$

$A_1(v,w,(m,n)) = \{v + w\}$

$R_2(v,(m,n)) = vertices(v)$ **and** $m \in v$ **and** $n \in v$

$A_2(v,(m,n)) = \{v\}$

Function *singleton* tests whether the multiset is a singleton or not. Reaction (R_1,A_1) consumes three elements of the multiset: two sets of vertices v and w and an edge connecting one element of v to one element of w. It yields a larger set of connected vertices $v + w$. Reaction (R_2,A_2) is just a specialization of (R_1,A_1) which involves only one vertex. It is used to remove edges of the graph that are no longer necessary. It is easy to see that if all the nodes of the graph are connected they will eventually be gathered into one set of vertices and all edges will be removed by (R_2,A_2). If the graph is not connected, the nodes will never be gathered into one set and the resulting multiset cannot be a singleton.

Shortest path

We consider now a weighted directed graph: a cost is associated with each edge and edges are "one-way". The length of a path is the sum of the costs of its edges. The problem consists in finding the length of the shortest path from x to y for all pairs of vertices (x,y). The initial multiset contains one triple (n,m,c) per pair of vertices (n,m) in the graph. If an edge (n,m) is present in the initial graph, then c is the cost of the edge (n,m), otherwise the value of c is ∞ (∞ satisfying the property $\forall n, n < \infty$). The resulting multiset is composed of triples (n,m,c) where c represents

the length of the shortest path from n to m. If there is no path from n to m the value of c is ∞. The GAMMA program is the following:

$$shortest\_path(G) = \Gamma(R,A))\ (G)\ \textbf{where}$$
$$R((v_1,v_2,c_{12}),(v_2,v_3,c_{23}),(v_1,v_3,c_{13})) = c_{13} > c_{12} + c_{23}$$
$$A((v_1,v_2,c_{12}),(v_2,v_3,c_{23}),(v_1,v_3,c_{13})) = \{(v_1,v_2,c_{12}),(v_2,v_3,c_{23}),(v_1,v_3,c_{12} + c_{23})\}$$

The cost associated with a pair of vertices represents the length of the shortest known path between the two vertices. The program performs a relaxation, decreasing costs until they represent the length of the shortest path.

Minimum spanning tree

We define the cost of a weighted graph as the sum of the weights of its edges. The problem is to find a minimum spanning tree which is defined as a subgraph of minimum cost connecting all the vertices of the graph. The idea for solving this problem in GAMMA is to proceed by reduction, building larger and larger local minimum spanning trees until all the vertices are included into one single tree which is a minimal spanning tree of the graph. Initially local minimum spanning trees are just individual vertices and two local spanning trees are aggregated using the shortest edge between them. We represent a graph as a multiset of quadruples (V,ST,E,M). V is a set of vertices connected by the edges in ST; the graph represented by the edges in ST is a minimum spanning tree connecting the vertices in V. E is the set of triples (n,m,c) such that $n \in V$ and there is an edge of cost c connecting n and m; M is a triple (n,m,c) of E of minimal cost such that $m \notin V$. The initial multiset contains one element $(\{n\},\emptyset,E,(n,m,c))$ per vertex n of the graph. The following GAMMA program computes the minimum spanning tree of a graph:

$$min\_st(G) = ST \quad \textbf{where} \quad \{(V,ST,E,M)\} = \Gamma((R,A))\ (G) \quad \textbf{where}$$
$$R((V_1,ST_1,E_1,(n_1,m_1,c_1)),(V_2,ST_2,E_2,(n_2,m_2,c_2))) = m_1 \in V_2$$
$$A\ ((V_1,ST_1,E_1,(n_1,m_1,c_1)),(V_2,ST_2,E_2,(n_2,m_2,c_2))) =$$
$$\{(V_1 + V_2,\ ST_1 + ST_2 + \{(n_1,m_1,c_1)\},\ E_1 + E_2,\ min(V_1+V_2, E_1+E_2)\}$$

$$min(V,E) = (n,m,c) \quad \textbf{where} \quad \{(n,m,c)\} = \Gamma((R'',A''))\ ((\Gamma((R',A'))\ (V + E)) - V) \quad \textbf{where}$$
$$R'(n,(n',n,c)) = True$$
$$A'(n,(n',n,c)) = \{n\}$$
$$R''((n_1,m_1,c_1),(n_2,m_2,c_2)) = c_1 \leq c_2$$
$$A''((n_1,m_1,c_1),(n_2,m_2,c_2)) = \{(n_1,m_1,c_1)\}$$

The program $min(V,E)$ performs a reduction: it eliminates edges connecting two elements of E (reaction (R',A')) and then removes from the multiset all edges but one (reaction (R'',A'')): the residual element of the multiset is an edge of minimal cost.

GEOMETRIC PROBLEMS

The problems we have studied so far have involved numbers, texts or graphs; we describe in this section two problems involving points: the convex hull problem and an image processing application. Both problems are handled by relaxation, starting with an approximation of the result.

Convex hull

The convex hull of a set of points in the plane is defined to be the smallest convex polygon containing them all. A convex polygon has the property that any line connecting two points inside the polygon must lie entirely inside the polygon [10]. It is easy to show that the vertices of the convex hull of a set of points P are elements of P. The GAMMA program is based on the following property: a point of the original set is a vertex of the convex hull if and only if it is not strictly inside a triangle made of three other points of the set. The initial multiset contains the coordinates of all the points and computation proceeds by throwing out points that fall inside a triangle.

$$convex(Points) = \Gamma((R,A))\ (Points) \quad \textbf{where}$$
$$R((i_1,j_1),(i_2,j_2),(i_3,j_3),(i_4,j_4)) = inside\ ((i_4,j_4),((i_1,j_1),(i_2,j_2),(i_3,j_3)))$$
$$A((i_1,j_1),(i_2,j_2),(i_3,j_3),(i_4,j_4)) = \{(i_1,j_1),(i_2,j_2),(i_3,j_3)\}$$

Function *inside* takes a point and a triangle and returns *true* if the point falls inside the triangle. It involves intricate tests and computations on the coordinates of the points. A generalization of this function to any polygon is given in [10].

An image processing application

A theory called mathematical morphology was proposed some years ago to solve problems in image processing applications [20]. We describe the treatment in GAMMA of edge detection, a classical image processing problem, along these lines. Each point of the image is originally associated with a grey intensity level; then an intensity gradient is computed at each point and edges are defined as the points where the gradient is greater than a given threshold T. The gradient at a point is computed relative to its neighbours: only points at a distance d less than D are considered for the computation of the gradient. The gradient at a point is defined in the following way:

$$G(P) = maximum(neighbourhood) - minimum(neighbourhood)$$
$$\textbf{where } neighbourhood = \{intensity(P') \mid distance(P,P') < D\}$$

Functions *maximum* and *minimum* yield respectively the maximum and the minimum of a multiset of values. The GAMMA program uses a multiset of quadruples (P,l,min,max); P is a pair representing the coordinates of a point, l is the intensity level of the point and *min* and *max* are the current values of *minimum(neighbourhood)* and *maximum(neighbourhood)* of the point. The initial value of *min* and *max* is l. The evaluation consists in decreasing *min* and increasing *max* until the limit values are reached. A second GAMMA program removes from the multiset the points where the gradient is less than the threshold.

$edges(Points) = select(\Gamma((R_1,A_1),(R_2,A_2))\ (Points))$ **where**

$R_1((P,l,min,max),(P',l',min',max')) = (distance(P,P') < D$ **and** $(l' < min)$

$A_1((P,l,min,max),(P',l',min',max')) = \{(P,l,l',max),(P',l',min',max')\}$

$R_2((P,l,min,max),(P',l',min',max')) = (distance(P,P') < D$ **and** $(l' > max)$

$A_2((P,l,min,max),(P',l',min',max')) = \{(P,l,min,l'),(P',l',min',max')\}$

$select(Points) = \Gamma((R_1,A_1),(R_2,A_2))\ (Points)$ **where**

$R_1((P,l,min,max)) = max - min < T$

$A_1((P,l,min,max)) = \{\}$

$R_1((P,l,min,max)) = max - min \geq T$

$A_1((P,l,min,max)) = \{(P,l)\}$

The treatment of a larger image processing application (namely the recognition of the tridimensional topography of the vascular cerebral network) in GAMMA is described in [7].

PROCESS SYNCHRONIZATION

In this section, we present solutions in GAMMA to some problems usually chosen in concurrent programming as typical challenges to test the expressiveness of synchronization constructs: the dining philosophers problem and a resource allocation problem. Before describing these programs let us emphasize that we use GAMMA in a quite different way in this section: we are not interested in the result of the evaluation (the problems we solve here typically require non terminating programs) but rather in the possible values of the multiset during the computation. In these examples, the multiset is seen as a representation of the state of the system. Furthermore, GAMMA being a very high level formalism, it makes it possible to express very concise solutions to these problems. Of course these solutions must rely on a correct implementation of the formalism (one possible implementation is described in [2]).

The dining philosophers

The formulation of the problem is the following: five philosophers spend their lives thinking and eating. They share a common dining room where there is a circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table there is a bowl of spaghetti which is endlessly replenished, and the table is laid with five forks. However the spaghetti is so hopelessly entangled that two forks are necessary simultaneously to eat.

The problem is to find a protocol ensuring two fundamental properties: (1) there is no deadlock in the system and (2) there is no starvation (any hungry philosopher will be able to eat after a finite amount of time). Each philosopher may only use his two adjacent forks, and may only eat for a finite amount of time.

The state of the system is represented by a multiset containing the available forks and the identities of eating philosophers. The initial multiset $\{F_0, F_1, F_2, F_3, F_4\}$ contains five forks; imagine that philosopher P_1 is allowed to eat with forks F_1 and F_2, then the new multiset will be: $\{F_0, P_1, F_3, F_4\}$. When P_1 has finished eating he returns his two forks so that another philosopher may use them. The following GAMMA program expresses this protocol ($\oplus$ represents addition modulo 5):

$philosophers = \Gamma((R_1, A_1)\ (R_2, A_2))\ (\{F_0, F_1, F_2, F_3, F_4\})$ **where**

$R_1\ (F_i, F_{i\oplus 1}) = True$

$A_1\ (F_i, F_{i\oplus 1}) = \{P_i\}$

$R_2\ (P_i) = True$

$A_2(P_i) = \{F_i, F_{i\oplus 1}\}$

The GAMMA paradigm allows a direct expression of a deadlock-free solution because the basic synchronization facility offered is precisely the *atomic operation* on a collection of items. The absence of starvation is guaranteed by the assumption of *fairness* in the selection process. A parallel implementation of GAMMA exhibiting these properties is described in [2].

Three solutions to this problem are presented in [3]. First, philosophers try to collect independently their left and right forks and this may lead to a deadlock situation because all philosophers may decide to take their left fork simultaneously ... before trying to seize their right forks. A second solution allows a philosopher to seize atomically two forks; this solution uses the monitor concept which offers nice synchronization facilities. However, this solution does not guarantee the absence of starvation because the philosophers can conspire in order to prevent one of their number from getting his two forks. This problem cannot be solved without introducing some asymmetry into the protocol. This can be done by allowing only four philosophers around the table at a given time.

A resource allocation problem

Consider a computing system where n users $U_1, ..., U_n$ share a common pool of resources $\{r_1, ..., r_k\}$. Each user can be in one of the three following states: *passive*, *waiting* for a resource and *busy*. Transitions between these three states can occur as follows: *passive* $\rightarrow$ *waiting* $\rightarrow$ *busy* $\rightarrow$ *passive* .... The state of the system is represented by a multiset containing the identities of free resources, the identities of busy users represented by pairs ($user_i$, *resource allocated*$_j$), the identities of waiting users represented by pairs ($user_i$, *waiting*) and the identities of passive users represented pairs ($user_i$, *passive*). The following GAMMA program solves the problem:

$resource\_management = \Gamma((R_1, A_1), (R_2, A_2), (R_3, A_3))\ (\{r_1, ..., r_k, (U_1, passive), ..., (U_n, passive)\})$

where

$R_1((U_i, passive)) = True$

$A_1((U_i, passive)) = \{(U_i, waiting)\}$

$R_2((U_i, waiting), r_j) = True$

$A_2((U_i, waiting), r_j) = \{(U_i, r_j)\}$

$R_3((U_i, r_j)) = True$

$A_3((U_i, r_j)) = \{(U_i, passive), r_j\}$

We use a single multiset to represent users and resources. It would be possible to extend GAMMA with appropriate syntactic sugar to be able to use a separate multiset for each user and for resources. This would allow us to keep the programming closer to the logic of the problem, but would not change the spirit of the solution.

CONCLUSION

We have shown in this paper that the multiset transformation paradigm can be used to provide elegant solutions to a wide range of programming problems. We have exhibited three basic programming principles (namely *relaxation*, *expansion* and *reduction*) that we have put into practice to build all the programs presented. It may be surprising at first glance that such a simple formalism possesses enough expressive power to solve in a natural way such different problems. We believe that the most fruitful approach to programming language design is to start with a few basic principles and to exploit them as far as possible. In this respect, we cannot resist the temptation to quote E.W.Dijkstra eighteen years ago: "*Another lesson we should have learned from the recent past is that the development of "richer" or "more powerful" programming languages was a mistake in the sense that these baroque monstrosities, these conglomerations of idiosyncrasies, are really unmanageable, both mechanically and mentally. I see a great future for very systematic and very modest programming languages.*" [8].

Program derivation

It should be clear that GAMMA is not a programming language in the usual sense of the term. GAMMA programs are executable but any straightforward implementation would be extremely inefficient. We see GAMMA as a convenient intermediate language between specifications and programs: it is possible to express in GAMMA the *idea* of an algorithm without any detail about the order of execution or the memory management. For example, you can tell in GAMMA that your strategy to sort a sequence is to exchange values, that you want to find the convex hull by removing the points inside a triangle, and so forth. Further program derivation may specialize these abstract programs by choosing a particular data representation and a particular execution order. For example, selection sort, bubble sort or quicksort can be obtained from the abstract exchange sort program presented in this paper by applying different derivation strategies. Actually GAMMA is currently used in the context of program derivation [1,2]: GAMMA programs are first derived from a specification in first order logic; then a second derivation step is performed to obtain traditional programs (for sequential or parallel machines) from this GAMMA program. The derivation is based on the notions of variant and invariant properties [9,13]. Using an intermediate language like GAMMA makes the derivation easier because it allows a nice separation of concerns: the first step (the derivation of the GAMMA program) is related to the logic of the algorithm, whereas the second step expresses lower level choices such as data representation or execution order. In order to show the relevance of GAMMA for program derivation, let us consider an informal specification of a sorting program:

$$
\begin{aligned}
& M = sort(M_0) \ <=> \\
& \qquad \forall (i,v),(j,w) \in M,\ i > j => v \geq w \quad \textit{and} \qquad (1) \\
& \qquad \{i \mid (i,v) \in M\} = \{1, \ldots, card(M_0)\} \quad \textit{and} \qquad (2) \\
& \qquad \{v \mid (i,v) \in M\} = M_0 \qquad (3)
\end{aligned}
$$

Let us choose *(1)* as the variant property: this implies that the program has to proceed while the negation of *(1)* holds:

$$not(1) = \exists (i,v), (j,w) \in M,\ i > j \textbf{ and } v < w$$

If we look at the definition of GAMMA, we can see that the reaction condition precisely states the condition to be satisfied by some elements of the set for the computation to proceed. So the reaction condition can be derived from the negation in a straightforward way:

$$R((i,v),(j,w)) = (i > j) \textbf{ and } (v < w)$$

According to the invariant *(2)* ***and*** *(3)*, the indexes and values cannot be modified; so the only possible action consists in exchanging values and indexes:

$$A((i,v),(j,w)) = \{(i,w),(j,v)\}$$

The termination of the derived program can be shown using a multiset ordering. The invariant can then be used to derive an array implementation of the multiset (the set of indexes is constant and equal to $\{1,...,n\}$); the most straightforward orders of execution lead to selection sort, insertion sort or bubble sort but quicksort or heapsort can be derived as well (the latter involving a different data representation choice).

Another remarkable benefit of using GAMMA in the derivation is that GAMMA programs do not have any sequential implementation bias. In fact it is often the case that problems which are usually considered as inherently sequential turn out to have a parallel solution in GAMMA (and very often this is indeed the most natural solution). The spanning tree problem, the set partitioning problem and the longest upsequence example are good illustrations of this property.

Related works

It is interesting to note that even in the context of program construction many people have felt the need to describe algorithms in an abstract way very much in the spirit of GAMMA. Let us quote for example Sedgewick in [19], page 489:"*The Ford-Fulkerson method described above can be summarized as follows: "start with zero flow everywhere and increase the flow along any path from source to sink with no full forward edges or empty backward edges, continuing until there are no such paths in the network." But this is not an algorithm in the usual sense, since the method for finding paths is not specified, and any path at all could be used.*" This is followed by the description of a particular implementation of this idea for an algorithm. In the same way Goldberg and Tarjan have recently presented a new algorithm for the maximum-flow problem by giving first an abstract version in the form of two *applicability conditions* and associated *actions* [12].

The term "*production systems*" has been used rather loosely in artificial intelligence to denote systems described in terms of a global database, a set of production rules and a control system [17]. The production rules operate on the global database. Rules are associated with applicability conditions and the control system chooses the next rule to apply. A global termination condition is used to stop the computation. The globality of the production rules and termination conditions and the specification of control make these systems rather different from the GAMMA

formalism. These systems include a control component because the need to express a control strategy is often crucial in artificial intelligence applications. However very restricted forms of production systems called *decomposable production systems* exhibit locality properties allowing certain freedom in the order of application of the rules [17].

Actually several formalisms bearing some similarities to GAMMA have been proposed recently, which seems to denote a current trend towards high level languages of this form. In [6], Chandy and Misra describe a language, called UNITY (for Unbounded Nondeterministic Iterative Transformations), and its associated proof system. A UNITY program is essentially a declaration of variables and a set of multiple-assignment statements. Program execution consists in selecting nondeterministically some assignment statement, executing it and repeating forever. Nondeterministic selection is constrained by the following "fairness" rule: every statement is selected infinitely often. The main objective of UNITY is the systematic development of programs which can be implemented on different (distributed or centralized) architectures. Program development is carried out in two basic steps: first a correct program is derived from specifications, then this program is adapted to the target architecture; this adaptation is achieved by transformations of the original program in order to make control explicit. The multiple-assignment statement is used to express the mapping onto synchronous shared-memory architectures and the mapping onto asynchronous architectures is achieved by the partitioning of the statements of the program.
The major differences between the GAMMA model and UNITY may be summarized as follows:

- UNITY is based on a static data structure, the array, which makes less natural the treatment of problems involving data whose size may evolve dynamically.

- the multiple assignment statement, which is the basis of UNITY, entails an imperative style of programming.

- the notion of locality is not emphasized as it is in GAMMA. Computations which may be carried out in parallel are determined in a special design phase which aims at mapping the UNITY program onto a particular target machine. This mapping phase transforms a UNITY program without explicit parallelism into an explicitly parallel program; this phase is carried out as rigorously as possible but still remains informal.

- the associated proof techniques are more complex and program derivation is more laborious especially when dealing explicitly with parallelism.

However we should mention that the goal of the proponents of UNITY was a bit different to ours since we do not attempt to model within the GAMMA formalism the execution of programs on various kinds of architectures (although GAMMA programs can also be mapped on various architectures).

A programming notation, called *associons*, has been proposed in [18]. Essentially, an associon is a tuple of names defining a relation between entities represented by these names. The state of the computation can be changed by the creation of new associons representing new relations deduced from the already existing ones. Such deductions are described in a closure statement whose execution may be decomposed into several simple activities which may be run in parallel.

The spirit of the proposal is quite similar to the ideas which have led to the GAMMA model. However several important differences may be highlighted:

- the locality principle which is of prime importance in the GAMMA formalism is not emphasized in the associons model. This comes essentially from the fact that negated presence conditions (which correspond to global properties on the set of tuples) are allowed in the associon model.

- unlike GAMMA, the associon model is deterministic; in order to ensure this property, the execution of an action (creation of new associons) cannot invalidate another action; this entails a potential independence between actions but introduces restrictions on the type of actions which are permitted.

- GAMMA is based on multisets while the associon model is based on sets. We find that the extra degree of freedom provided by the use of multisets is very useful as far as program construction is concerned. Furthermore, this freedom is necessary to satisfy the locality property (no global test is needed to check that a produced element is not already present).

Let us also mention the Linda approach [5,11]. Linda contains a few simple commands operating on a tuple space. Adding these tuple-space commands to an existing base language produces a parallel programming dialect. Linda's model is based on generative communications. If two processes need to communicate, the producer adds a tuple to a particular domain, and the consumer may read (destructively or not) this information from the tuple space. Data and program objects are represented in a uniform way as passive or active tuples. Of course, several processes may be active on the same tuple space, thus allowing parallel tuple processing. Linda is a very simple communication model that can easily be incorporated into existing programming languages. As such, Linda is not a computational model. However, in the same way as the GAMMA model, it shows clearly how advanced data structuring facilities such as tuple spaces or multisets may greatly simplify the programming task.

Last but not least, let us mention the Chemical Abstract Machine (or *cham*) proposed by Berry and Boudol [4] to model asynchronous concurrent computations. The *cham* is an elaboration on the original GAMMA formalism introducing the notion of subsolution enclosed in a membrane. It is shown that models of algebraic process calculi can be defined in a very natural way using a *cham*. The fact that concurrency is the primitive built-in notion makes proofs far easier than in the usual process semantics.

REFERENCES

1. Banâtre, J.-P., and Le Métayer D. A new computational model and its discipline of programming. INRIA Research Report 566 (Sept. 1986).
2. Banâtre, J.-P., Coutant, A., and Le Métayer, D. A Parallel Machine for Multiset Transformation and its Programming Style. *Future Generation Computer Systems.* 4, (1988), 133-144.
3. Ben-Ari, M. *Principles of concurrent programming.* Prentice/Hall International, 1982.
4. Berry, G., and Boudol, G. The Chemical Abstract Machine. In *Proceedings of ACM Symp. on Principles of Programming Languages* (San Francisco, Calif.). ACM, New York, 1990, pp. 81-94.
5. Carriero, N., and Gelernter, D. Linda in Context. *Commun. ACM* 32, 4 (April 1989), 444-458.
6. Chandy, M., and Misra, J. *Parallel Program Design: a Foundation.* Addison-Wesley Publishing Company, 1988.
7. Creveuil, C., and Moguerou, G. Dérivation d'un algorithme de segmentation d'images: un exemple d' application du formalisme GAMMA. INRIA Research Report 1049 (June 1989).
8. Dijkstra, E. W. The humble programmer. *Commun. ACM* 15, 10 (October 1972), 859-866.
9. Dijkstra, E. W. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, N.J., 1976.
10. Dromey, R. G. *Program Derivation.* Addison-Wesley Publishing Company, International Computer Science Series, 1989.
11. Gelernter, D. Generative Communication in Linda. *ACM Trans. Prog. Lang. Syst.* 7, 1 (Jan. 1985), 80-112.
12. Goldberg, A. V., and Tarjan, R. E. A new approach to the maximum-flow problem. *Journal of ACM* 35, 4, (Oct. 1988), 921-940.
13. Gries, D. *The Science of Programming.* Springer Verlag, New York, 1981.
14. Gries, D. A hands-in-the-pocket presentation of a k-majority vote algorithm, in *Formal Development of Programs and Proofs*, ed. E. W. Dijkstra (University of Texas at Austin Year of Programming series). Addison-Wesley Publishing Company, 1990, pp. 43-46.
15. Hoare, C. A. R. Communicating Sequential Processes, *Commun. ACM* 21, 8 (Aug. 1978), 666-677.
16. Knuth, D. *Seminumerical Algoritms. The Art of Computer Programming.* Addison-Wesley Publishing Company, 1969.
17. Nilsson, N. J. *Principles of Artificial Intelligence.* Tioga publishing company, Palo Alto, 1980.
18. Rem, M. Associons: A Program Notation with Tuples instead of Variables. *ACM Trans. Prog. Lang. Syst.* 3, 3 (July 1981), 251-262.
19. Sedgewick, R. *Algorithms.* Addison-Wesley Publishing Company, 1988.
20. Serra, J. *Image analysis and mathematical morphology.* Academic Press, 1982.

Liste des dernières publications internes parues à l'IRISA

PI 516 **COMMENT INTRODUIRE LA CONTIGUITE EN ANALYSE DES CORRESPONDANCES ? Application en segmentation d'image.**
Brigitte ESCOFIER, Habib BENALI, Kaddour BACHAR
Février 1990, 26 Pages.

PI 517 **MACHINE MODELING AND LOOP OPTIMIZATION FOR HORIZONTAL MICROCODED MACHINES**
François BODIN, François CHAROT
Février 1990, 24 Pages.

PI 518 **MULTISCALE SYSTEM THEORY**
Albert BENVENISTE, Ramine Nikoukhah, Alan S. Willsky.
Février 1990, 30 Pages.

PI 519 **PANDORE : A SYSTEM TO MANAGE DATA DISTRIBUTION**
Françoise ANDRE, Jean-Louis PAZAT, Henry THOMAS
Février 1990, 14 Pages.

PI 520 **SCHEDULING AFFINE PARAMETERIZED RECURRENCES BY MEANS OF VARIABLE DEPENDENT TIMING FUNCTIONS**
Christophe MAURAS, Patrice QUINTON
Sanjay RAJOPADHYE, Yannick SAOUTER
Février 1990, 14 Pages.

PI 521 **COMPUTABILITY OF RECURRENCE EQUATIONS**
Yannick SAOUTER, Patrice QUINTON
Février 1990, 28 Pages.

PI 522 **PROGRAMMING BY MULTISET TRANSFORMATION**
Jean-Pierre BANATRE, Daniel LE METAYER
Mars 1990, 26 Pages.

PI 523 **GOTHIC MEMORY MANAGEMENT : A MULTIPROCESSOR SHARED SINGLE LEVEL STORE**
Béatrice MICHEL
Mars 1990, 20 Pages.

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249-6399