View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by INRIA a CCSD electronic archive server

A new computational model and its discipline of programming

Jean-Pierre Banâtre, Daniel Le Métayer

▶ **To cite this version:**

Jean-Pierre Banâtre, Daniel Le Métayer. A new computational model and its discipline of programming. [Research Report] RR-0566, Inria. 1986. inria-00075988

HAL Id: inria-00075988

https://hal.inria.fr/inria-00075988

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

CENTRE DE RENNES
IRISA

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° 566

A NEW COMPUTATIONAL MODEL AND ITS DISCIPLINE OF PROGRAMMING

Jean-Pierre BANATRE
Daniel LE METAYER

Septembre 1986

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTÈMES ALÉATOIRES

Campus Universitaire de Beaulieu
Avenue du Général Leclerc
35042 - RENNES CÉDEX
FRANCE
Tél. : (99) 36.20.00
Télex : UNIRISA 95 0473 F

A NEW COMPUTATIONAL MODEL AND ITS DISCIPLINE OF PROGRAMMING

Γ : UN FORMALISME POUR LA CONSTRUCTION ET LA PREUVE DE PROGRAMMES

Jean-Pierre BANATRE, Daniel LE METAYER
IRISA/INRIA
Campus de Beaulieu
35042 RENNES CEDEX
FRANCE

Publication Interne n° 305
Juillet 1986
24 Pages

Abstract:

The objective of the work described in this paper is to take advantage of the functional programming style in order to mechanize reasoning about programs. The underlying idea comes from the observation of the weaknesses of functional languages, as far as program reasoning is concerned, when dealing with recursive functions. Our approach advocates the use of a high level combinator which makes explicit recursion unnecessary, thereby simplifying the reasoning about programs. This combinator relies on the chemical reaction metaphor: the only data structure is the multiset and the computation can be seen as a succession of chemical reactions consuming elements of the multiset and producing new elements according to particular rules. The style of programming implied by this new combinator is illustrated by some examples. Particular emphasis is put on fact that this computational model provides a good basis for program synthesis. Furthermore we describe a small set of rules which can be used to derive significant properties of realistic programs. A mechanization of the method is presented and the proofs of some non trivial programs, such as the exchange-sort or the sieve of Eratosthenes, are detailed.

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)

Résumé:

L'objectif du travail décrit dans cet article consiste à exploiter les avantages des langages fonctionnels pour mécaniser les raisonnements sur les programmes. Nous partons de la constatation que l'utilisation de combinateurs de haut niveau facilite les raisonnements sur les programmes. Nous proposons donc un combinateur qui permet d'écrire des programmes sans récursivité explicite. Ce combinateur repose sur la métaphore de la réaction chimique: la seule structure de données est le multiensemble et l'exécution peut être vue comme une suite de réactions chimiques consommant des éléments de l'ensemble et produisant de nouveaux éléments selon certaines règles. Nous illustrons par quelques exemples le style de programmation impliqué par ce combinateur. Nous insistons sur le fait que ce formalisme peut servir de base à un système de synthèse de programmes. De plus, nous décrivons un petit ensemble de règles qui sont utilisées pour dériver des propriétés des programmes. Nous montrons comment cette dérivation peut être mécanisée et nous présentons la preuve de programmes non triviaux comme le crible d'Eratosthènes et le tri par échanges.

1. INTRODUCTION.

Programming languages and programming environments have attracted much attention in the last few years. It is now recognized that further progress in these fields rely on the ability to define the semantics of programming languages in a concise and natural way and to design useful software tools based on this semantics [3]; these tools should include semantic editors [7], program transformation systems [4, 5] and program complexity evaluators [9].

Until now only functional languages seem to fulfil partially these requirements; for example all realistic program transformation systems are based on a functional programming language [4, 5]. The FP style of programming [1] is particularly well suited to mechanical program manipulation [9, 10] mainly because of the structured programming implied by its high level functional forms. These few functional forms express very common recursive patterns; for example:

$(\alpha\, f): <x_1, \dots, x_n> = <f{:}x_1, \dots, f{:}x_n>$

$(/\, f): <x_1, \dots, x_n> = f{:}<x_1, f{:}<x_2, \dots f{:}<x_{n-1}, x_n> \dots >>$

Some programs can be written in a very elegant and concise way using these functional forms; for example the program evaluating the length of a sequence can be written:

length = (/ +) o (α "1")

This definition is so obvious that it could almost be considered as a specification of the program: the length of a sequence is obtained by counting one for every element and adding all these numbers.

Furthermore a very powerful set of axioms is associated with these functional forms, allowing the formal proof of some non trivial properties; for example the

following axioms are valid within FP semantics:

$$(\alpha\ f) \text{ o apndl o } [g, h] = \text{apndl o } [f \text{ o } g, (\alpha\ f) \text{ o } h] \quad \text{(A1)}$$

$$(/f) \text{ o apndl o } [g, h] = f \text{ o } [g, (/f) \text{ o } h] \quad \text{(A2)}$$

where apndl is a primitive function defined by:

$$\text{apndl} : \langle x_1, \langle x_2, \ldots, x_n\rangle\rangle = \langle x_1, \ldots, x_n\rangle$$

We can use these axioms to prove that:

$$\text{length o apndl o } [f, g] = + \text{ o } [\text{"1"}, \text{length o } g]$$

in the following way:

length o apndl o [f, g] = /+ o (α "1") o apndl o [f, g]	by definition
= /+ o apndl o ["1" o f, ((α "1") o g)]	by A1
= /+ o apndl o ["1", ((α "1") o g)]	property of constants
= + o ["1", /+ o ((α "1") o g)]	by A2
= + o ["1", (/+ o (α "1")) o g]	associativity of "o"
= + o ["1", length o g]	by definition

So the proof of properties of FP programs are possible when these programs are expressed in terms of the functional forms of the language, whithout explicitly using recursion. However FP combinators are not powerful enough, so that recursion is generally needed in order to design realistic programs, thus making reasoning about programs far more difficult.

Another reason why the FP formalism is not well suited to the proof of real programs is that FP is a first order language, that is to say it is not possible to define higher order functions in FP. The basic principle underlying the proof of properties in FP is to express the property in the language itself and to prove it by program transformation. So the limitations of the language entail strong restrictions on the class of properties that we can prove in FP. It is quite hard for example to achieve the proof of a sorting program as the property itself is very difficult to write in FP. For example, the program expressing the fact that a sequence is well ordered can be written:

ordered = null -> "T" ; null o tl -> "T" ; > o [1, 2] -> "F" ; ordered o tl

This is a recursive definition which is not well suited to a proof in FP; a non recursive definition would be even more untractable:

ordered = (/ and) o ($\alpha \leq$) o trans o [tlleft, tl]

tl, tlleft, and trans being the following primitive functions:

tl: $<x_1, \dots, x_n> = <x_2, \dots, x_n>$

tlleft: $<x_1, \dots, x_n> = <x_1, \dots, x_{n-1}>$

trans: $<<x_{11}, \dots, x_{1n}>, \dots, <x_{m1}, \dots, x_{mn}>> =$

$<<x_{11}, \dots, x_{m1}>, \dots, <x_{1n}, \dots, x_{mn}>>$

To prove the correctness of a sorting program S we have to show that:

ordered o S = "T"

which is quite difficult with any of the two definitions of ordered given above.

In order to overcome these difficulties we propose a new computational model expressed by a combinator, called Γ, which can be used to define in a natural way a large class of common programs. Furthermore the rules associated with this combinator allow the proof of interesting properties of non trivial programs such as sorting programs for example. Section 2 gives an informal definition of Γ and put emphasis on the systematic construction of Γ-programs. A more formal description of the Γ-system is given in section 3. Section 4 proposes a method which generalizes formal derivation techniques for conventional programs and makes it possible to derive formal properties of programs written in the Γ-system. Furthermore, a mechanization of the method is described. Section 5 contains a brief review and discussion.

2. SYSTEMATIC CONSTRUCTION OF PROGRAMS.

The computational model underlying the Γ combinator is based on the chemical reaction metaphor; we consider the data as a set of molecules and the computation is a succession of chemical reactions according to particular rules. These rules indicate which kind of molecules can react together and what these reactions produce; they are represented by the arguments of Γ:

$$\Gamma((R_1, A_1), \ldots, (R_m, A_m))$$

The R_i functions are called "reaction conditions"; they are boolean functions indicating in which case some elements of the multiset can react. The A_i functions ("actions") describe the result of these reactions. We should point out that several

reactions may be possible at the same time. No assumption is made on the order in which these reactions are actually achieved; we only impose that, when the reaction condition holds for at least one subset of elements, at least one reaction occurs; this means that the computation does not stop until the reaction condition does not hold for any subset of the multiset. This feature provides us with a very interesting property of the result which is useful to prove the partial correctness of the program.

Let us now take a small example showing the interest of this computational model with regard to program construction and program proof. We describe the construction of a sorting program in a systematic way to suggest how this model can be used for semi-automatic program synthesis.

We first have to define the representation of the data as a multiset. The natural choice consists in representing a value "v" whose index is "i" as a couple (v, i). Let M be any multiset of couples (v, i); we denote by M.v and M.i respectively the multiset of the values of elements of M and the multiset of the indexes of elements of M.

Then we have to give the specification of the program; let M_0 be the initial multiset and M_r the result of the computation. The specification of a sorting program could be:

$$M_r.v = M_0.v \qquad \text{(S1)}$$

$$M_r.i = \{1, \dots, card(M_0)\} \qquad \text{(S2)}$$

$$\forall\, e_1, e_2 \in M_r \quad e_1.i > e_2.i \Rightarrow e_1.v > e_2.v \qquad \text{(S3)}$$

S1 expresses the fact that the values of the result are exactly the values of the argument; S2 ensures that two different elements have two different indexes and that all indexes are comprised between 1 and card(M_0); S3 is the well ordering property (we assume that all values of the initial multiset are different).

The next step is the definition of the initial state. Actually this choice is connected with the definition of the general strategy of the algorithm; the idea consists in splitting the set of specifications in two parts:

- the invariant property which will be true at every step of computation; in particular this property will hold at the initial and the final steps;
- the variant property which will only be true at the last step of computation.

The invariant property gives a restriction on the initial state and the action while the variant property is used to find the reaction condition and the action. The reaction condition must express the fact that the variant property is not yet satisfied (so there is yet something to do) and the action must transform the multiset "in the right direction" (so as to reach a state verifying the variant property in a finite number of steps).

A possible choice in our example could be to take (S1 and S2) as the invariant property and S3 as the variant property; this means that the multiset of values must be preserved throughout the computation and that indexes must always be unique and comprised between 1 and card(M_0). Any initial state verifying these two properties is valid; so we can associate any index with each value, provided that it is unique and between 1 and card(M_0). The reaction condition must express the fact that S3 does not hold, that is to say that the sequence is not well ordered:

$$\exists\, e_1, e_2 \in M_r \quad (e_1.i > e_2.i) \text{ and } (e_1.v < e_2.v)$$

In our formalism the condition relation would be:

$$R(e_1, e_2) = (e_1.i > e_2.i) \text{ and } (e_1.v < e_2.v)$$

The next step is to find the action to transform the multiset "in the right direction" while preserving the invariant property. The invariant property tells us that we cannot remove or add any value or index, so we can only exchange them; in our formalism this would be written:

$$A(e_1, e_2) = \{(e_1.v, e_2.i), (e_2.v, e_1.i)\}$$

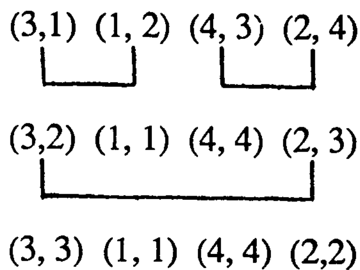
This means that the elements $(e_1.v, e_1.i)$ and $(e_2.v, e_2.i)$ of the multiset are replaced by the elements $(e_1.v, e_2.i)$, $(e_2.v, e_1.i)$. In order to prove that this action transforms the multiset "in the right direction", that is to say that computation terminates, we have to find a positive function of the state which is decreased throughout the computation. In our case we just have to remark that the reaction condition does not hold for the elements produced by the action. Actually it is easy to show that the total number of couples of elements verifying the reaction condition is a strictly decreasing function of the state. So action A fulfil all the requirements and we have the following sorting program:

sort = $\Gamma(R, A)$ <u>with</u>

$R(e_1, e_2) = (e_1.i > e_2.i)$ and $(e_1.v < e_2.v)$

$A(e_1, e_2) = \{(e_1.v, e_2.i), (e_2.v, e_1.i)\}$

Furthermore we know, by construction, that our program is correct.
Let us now describe the execution of this program applied to the sequence <3,1,4, 2>. This sequence is represented by the multiset {(3,1), (1,2), (4,3), (2,4)} which is the initial state. Two reacting elements are linked by an horizontal line:



The stable state is reached as no couple of elements verify the reaction condition; so the result is the sequence <1, 2, 3, 4>. Of course this is only one of the possible sets of configurations leading to the result.

This program can be seen as a generalized form of the exchange sort algorithm as any couple of ill-ordered elements can exchange their positions at any time. Many other choices could have been made during the construction of this program yielding very different algorithms. We do not want to go into further details on program synthesis here as the purpose is only to suggest the discipline of programming entailed by this model. As automatic program synthesis will not be possible in all cases, an environment for the Γ-system should include a tool for the proof of properties of programs. We give in next sections a more formal description of the Γ-system and its associated derivation technique.

3. FORMAL DEFINITION OF THE SYSTEM.

3.1. Data structures.

The basic information structuring facility is the multiset which is the same as a set except that it may contain multiple occurrences of the same element. Atomic components of multisets may be of type real, character, integer, multisetof type ... Any type t is associated with a boolean function t yielding the value true if its argument is of type t; for example integer:3 = T. We introduce a new type, called

index, whose elements are represented by relief integers; index elements are used for the construction of sequences which represent particular indexed multisets; an indexed multiset is a multiset composed of multisets containing one and only one element of type index. For example S = {{1,x},{2,y}} is an indexed multiset. From now on we shall call "sequence" any indexed multiset M with a set of indexes equal to {1, ..., card(M)}; for example the above multiset S can be written: <x,y>.

3.2. primitive functions.

Apart from the traditionnal operations on multisets ($\cup$, $\cap$, -, card, empty) we define the extraction primitive which is denoted by:

b: S = x if x is the unique element of the multiset S such that b:x =true.

b: S = $\perp$ otherwise.

This function allows the extraction of an element verifying a certain condition; for example positive:{-3, 1, 2} = $\perp$, index:{5, 2, 4} = 4.

Given the definition valof = not o index, the function valof yields the unique element of a multiset which is not an index; for example valof: {5, 2, 4} = 5.

We assume that all primitive functions of the FP system are available in the Γ-system (for example 1:<x, y> = x but if M is not a sequence then 1:M = $\perp$). FP primitives whose application to a multiset is meaningful (basically symmetric and associative primitives) are also available on multisets: for example + : {2,5} = 7, distl : <1,{3,4}> = {<1,3>,<1,4>}. For a detailed description of the FP language the reader should refer to [1].

3.3. combinators.

The Γ-system contains four combinators:

(1) composition.

$$(f \circ g): x = f : (g : x)$$

(2) multiset construction.

$$\{f_1, \dots, f_n\}: x = \{f_1{:}x, \dots, f_n{:}x\}$$

As no confusion is possible we use the same symbols for the multiset constructor and the multiset denotation ({}).

(3) constant.

$$"x" : y = x$$

(4) Γ combinator.

$\Gamma((R_1,A_1), \dots, (R_m,A_m)){:}M =$

$\exists\, x_1, \dots, x_{n1} \in M$ such that $R_1{:} \langle x_1, \dots, x_{n1}\rangle \rightarrow$

$\Gamma((R_1,A_1), \dots, (R_m,A_m)){:}(M - \{x_1, \dots, x_{n1}\}) \cup A_1{:}\langle x_1, \dots, x_{n1}\rangle;$

...

$\exists\, x_1, \dots, x_{nm} \in M$ such that $R_m{:} \langle x_1, \dots, x_{nm}\rangle \rightarrow$

$\Gamma((R_1,A_1), \dots, (R_m,A_m)){:}(M - \{x_1, \dots, x_{nm}\}) \cup A_m{:}\langle x_1, \dots, x_{nm}\rangle;$

otherwise M.

Let us point out that if several R_i hold at the same time, the choice which is made among them is not deterministic. However appropriate restrictions on the definition of actions A_i may ensure determinacy; this point is not developed here. The recursive combinators of FP (/, α) can be defined from the above four combinators (some examples are given in next section).

This is the most general definition of Γ; however most common programs can be expressed with m = 1, so we will restrict our discussion to this particular case and set $R = R_1$ and $A = A_1$. From now on we shall call R the reaction condition.

3.4. Examples of Γ-programs.

Let us now take some examples to illustrate the programming style entailed by the combinator Γ. We use extended definitions proposed by Backus [2] to describe functions as they provide a concise way to specify the arity of functions. We just give an informal description of this notation here; let d be the following FP function:

d = eq o ["2", length] -> squareroot o + o [* o [1, 1], * o [2, 2]] ; ⊥

A definition of d using extended definitions would be:

d o [X1, X2] = squareroot o + o [* o [X1, X1], * o [X2, X2]]

Example 1.

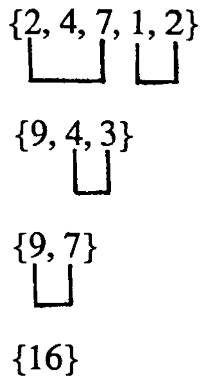
The tree combinator proposed by Williams [11] can be defined in the following way:

tree f = Γ(R,A) with

R o [X1,X2] = "T"

A o [X1,X2] = {f o [X1, X2]}

The following figure describes the computation of (tree +):<2,4,7,1,2>:



Of course this is again one among the possible paths leading to the stable state. This combinator is slightly more general than the original one presented in [11], as it operates on sets rather than sequences; so elements can be combined in many ways.

Example 2.

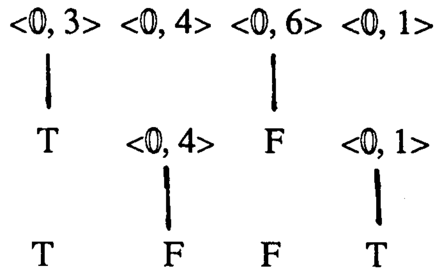
This is the definition of the combinator α on multisets:

$\alpha\, f = \Gamma(R,A) \text{ o distl o } [\text{"0"}, id]$ with

$R \text{ o } [X] = \text{index o } 1 \text{ o } X$

$A \text{ o } [X] = \{f \text{ o } 2 \text{ o } X\}$

In this case indexes are used as tags indicating that an element has not yet been dealt with. The following figure describes one possible evolution of the state during the evaluation of (α odd) : <3, 4, 6, 1>;



As the reaction condition is unary in this case all the computations could have been done in a single step.

Example 3.

Let us come back to the sorting problem. We want to sort a sequence $<x_1, ..., x_n>$ of different integers represented by an indexed multiset $\{\{1, x_1\}, ..., \{n,x_n\}\}$. So we want for any couple ($Xi = \{i,x_i\}$, $Xj = \{j, x_j\}$) of the multiset the following relationship:

$i<j \Rightarrow x_i < x_j$ that is to say

< o [index o Xi, index o Xj] ⇒ < o [valof o Xi, valof o Xj]

A straightforward solution can be:

sort = Γ(R,A) with

R o [X1,X2] = and o [< o [index o X1, index o X2], > o [valof o X1, valof o X2]]

A o [X1,X2] = {{index o X1, valof o X2}, {index o X2, valof o X1}}

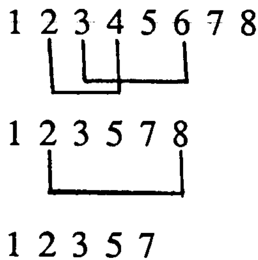
R expresses the fact that two elements are ill-ordered and A exchanges the indexes... so these two elements will be well ordered and so on and so forth till the relation R is false for any couple (x_i, x_j) which means that the whole set is well ordered.

Example 4.

The sieve of Eratosthenes can be written in a concise an elegant way using Γ:

sieve = Γ(R,A) o iota_m with
R o [X1, X2] = multiple o [X1, X2]
A o [X1, X2] = {X2}

where multiple o [X1, X2] is true if and only if X1 is a multiple of X2 ($X2 \neq 1$) and iota_m is defined by iota_m : n = {1, ..., n}.
One possible computation of sieve:8 may be figured as:



4. MECHANIZATION OF PROOF OF PROPERTIES.

The power of the Γ-style of programming has still to be further explored in order to assess its full capability; in order to limit our discussion let us only point out that when computation terminates, R is false for any sequence of elements of the multiset; this property can be stated in the following way:

$\nexists x_1, ..., x_n \in \Gamma(R,A) : M$ such that $R: \langle x_1, ..., x_n \rangle$

This information is an interesting property of the result which is directly available from the body of the program. For example, one can deduce that if the sieve program terminates, then the resulting multiset will not contain any elements x_i, x_j such that x_i is a multiple of x_j. In order to check the (partial) correctness of the program we have to prove that (1) all elements of the resulting multiset belong to the initial multiset and that (2) no prime number has been eliminated. Properties (1) and (2) are called invariance properties [6, 8]. We are now going to describe a method for the mechanical evaluation of Γ-program invariance properties.

4.1. Mechanical evaluation of invariance properties.

In order to mechanize the evaluation of invariance properties, we start with the following remark: during a computation step of Γ(R,A) (application of action A to a sequence of elements of the multiset) three kinds of events may occur: some elements of the multiset may disappear, some elements may be kept in the multiset and some elements may be created. So invariance properties that we would like to find can be classified into two main categories: conservation properties and generation properties. For the sake of briefness we shall restrict ourselves to the description of the conservation properties which are sufficient to treat the examples of this paper.

Notations.

we denote by ε a generalization of the $\in$ relation:

$x \varepsilon M \Leftrightarrow x \in M$ or $x \varepsilon M'$ with $M' \in M$

Let us now consider the program $P = \Gamma(R,A)$ with R and A being functions of arity n and $A{:}\langle x_1, \ldots, x_n\rangle = \{A_1{:}\langle x_1, \ldots, x_n\rangle, \ldots, A_m{:}\langle x_1, \ldots, x_n\rangle\}$.

The relation C_{RA} is defined as follows:

$$C_{RA}(x,M) = \forall s \ (((\alpha\ \underline{valof}) : s \subset M \text{ and } x \varepsilon s \text{ and } R{:}s) \Rightarrow x \varepsilon A{:}s)$$

The function (α valof) is used to translate a sequence into a multiset. $C_{RA}(x,M)$ is true if and only if x cannot be eliminated from M in one step of computation: either x cannot be involved in a transformation or x belongs to the result of such a transformation.

We say that C_{RA} is preserved if:

$$C_{RA}(x, M) \text{ and } M' = (M - \{x_1, \dots, x_n\}) \cup A{:} <x_1, \dots, x_n> \text{ and } R{:}<x_1, \dots, x_n> = T$$
$$\Rightarrow \quad C_{RA}(x,M')$$

This means that if the property C_{RA} holds for a multiset M and an element x of M, then it holds for any multiset M' produced from M using $\Gamma(R,A)$. In other words C_{RA} is preserved through the computation of $\Gamma(R,A)$.
The most general property that we can state about the conservation of elements is the following one:

Conservation property:
(CP) $(x \in M \text{ and } C_{RA}(x,M) \text{ and } C_{RA} \text{ is preserved}) \Rightarrow x \in \Gamma(R,A) : M$

This means that if x cannot be eliminated from M in one step of computation and cannot be eliminated from any M' produced from M (using $\Gamma(R,A)$) then x will necessarily be present in $\Gamma(R,A)$: M. This property is too general to be completely

mechanized. So we have worked out three weaker properties which can be automatically verified.

Rearrangement property.

(RP) $(\alpha f) \circ A = (\alpha f) \circ \alpha\, \underline{valof} \Rightarrow (\alpha f) \circ \Gamma(R,A) = \alpha f$

This property can be syntactically verified when the result of the action A is a rearrangement of its arguments.

Elimination properties.

(EP1) $(A{:}\langle x_1, \dots, x_n\rangle \subset \{x_1, \dots, x_n\}) \Rightarrow \Gamma(R,A){:}M \subset M$

(EP2) $(A{:}\langle x_1, \dots, x_n\rangle \subset \{x_1, \dots, x_n\}$ and $x \in M$ and

$\forall x_1, \dots, x_{n-1} \in M \; (R : \langle x_1, \dots x, \dots, x_{n-1}\rangle \Rightarrow x \in A : \langle x_1, \dots x, \dots x_{n-1}\rangle))$

$\Rightarrow \quad x \in \Gamma(R,A){:}\, M$

Property EP2 states that, if the result of A is included into the set of its arguments, an element of M which cannot be eliminated in the first step will never be eliminated. Relation $A{:}\langle x_1, \dots, x_n\rangle \subset \{x_1, \dots, x_n\}$ can be established by a static analysis of the body of A.

4.2. Some examples of proofs.

Let us come back to the examples of section 2.4. to illustrate the use of our rules. The exchange-sort program was defined by (Example 3):

sort = Γ(R,A) <u>with</u>

R o [X1,X2] = and o [< o [<u>index</u> o X1, <u>index</u> o X2], > o [<u>valof</u> o X1, <u>valof</u> o X2]]

A o [X1,X2] = {A1 o [X1, X2], A2 o [X1, X2]}

<u>with</u>

A1 o [X1,X2] = {<u>index</u> o X1, <u>valof</u> o X2}

A2 o [X1,X2] = {<u>index</u> o X2, <u>valof</u> o X1}

It is clear that EP1 and EP2 do not hold as:

$$A \circ [X1, X2] \not\subset \{X1, X2\}$$

On the other hand a static analysis of the body of A shows that:

<u>index</u> o X1 $\in$ A1 o [X1, X2] <u>index</u> o X2 $\in$ A2 o [X1, X2] and

<u>valof</u> o X1 $\in$ A2 o [X1, X2] <u>valof</u> o X2 $\in$ A1 o [X1, X2]

As A1 o [X1,X2] and A2 o [X1,X2] contain only two elements, one of which being of type index the following properties hold:

<u>index</u> o X1 = <u>index</u> o A1 o [X1, X2]

<u>index</u> o X2 = <u>index</u> o A2 o [X1, X2]

<u>valof</u> o X1 = <u>valof</u> o A2 o [X1, X2]

<u>valof</u> o X2 = <u>valof</u> o A1 o [X1, X2]

So we have:

$$(\alpha \ \underline{index}) \ o \ A \ o \ [X1,X2] = (\alpha \ \underline{index}) \ o \ \{X1,X2\} \quad \text{and}$$
$$(\alpha \ \underline{valof}) \ o \ A \ o \ [X1,X2] = (\alpha \ \underline{valof}) \ o \ \{X1,X2\}$$

So the rearrangement properties hold and we can conclude that:

$$(\alpha \ \underline{index}) \ o \ \Gamma(R, A) = (\alpha \ \underline{index}) \quad \text{and}$$
$$(\alpha \ \underline{valof}) \ o \ \Gamma(R,A) = (\alpha \ \underline{valof})$$

So we know that the set of indexes of the result is the same as the set of indexes of the argument and that the values are preserved as well. Furthermore the reaction condition does not hold for any couple of elements of the result, which means that the sequence is well ordered. These properties are sufficient to prove the (partial) correctness of the sort program.

The sieve of Eratosthenes has been defined by (Example 4):

sieve = Γ(R,A) o iota_m   <u>with</u>
  R o [X1, X2] = multiple o [X1, X2]
  A o [X1, X2] = {X2}

A straightforward analysis of the body of A shows that :

$$A \ o \ [X1, X2] \subset \{X1, X2\}$$

So the elimination properties EP1 and EP2 hold and we have:

$\Gamma(R,A):M \subset M$ and

$x \in M$ and $\nexists x_2 \in M$ such that multiple : $<x, x_2> \Rightarrow x \in \Gamma(R,A): M$

This property means that the result is a subset of the argument and that a number which is not a multiple of any other number can not be eliminated. So we know that the set of prime numbers of the argument is included in the result. Furthermore we know that there are no non-prime numbers in the result (as multiple : $<x_1, x_2>$ is false for any couple (x_1, x_2) of elements of the result); so the result is exactly the set of prime numbers of the argument. We can see that the partial correctness of the program is again a straightforward consequence of the the invariance property and of the falseness of the reaction condition.

5. CONCLUSION.

The Γ-system is currently under development as an extension of an already existing functional environment. As the Γ-system is a very high level formalism, optimization techniques have to be designed so as to produce a reasonably efficient implementation. Further work has still to be done in this area.

For the sake of briefness only the main ideas underlying our work have been described. In particular the problem of the termination of Γ-programs has not been tackled here. Let us only point out that the termination proof is straightforward when properties EP1 and EP2 hold, as the size of the multiset argument is decreasing through the computation. In general we have to exhibit a monotonic decreasing function from the reaction condition and the body of the action.

However the properties defined (rearrangement property and elimination properties) are often used to derive significant invariants. Furthermore we should mention that, even in cases where the invariance property we can synthetize is weak,

the falseness of the reaction condition does provide a relevant property.

Another promising area of research concerns the mechanization of the construction of Γ-programs. Examples of the sort program and the sieve program show that the body of the programs are quite close to the specification of these problems (Section 2.4.) and might be derived from these specifications. Current work on this problem seems to confirm this belief; the idea consists in deriving Γ-programs by dividing systematically the specification in two parts: the invariant part which is used to generate the action and the variant part which allows us to find the reaction condition. This method has already allowed us to produce some original algorithms.

References.

1. Backus, J. W. Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs. Commun. ACM 21, 8 (Aug. 1978), 613-641.
2. Backus, J. W. The algebra of functional programs: function level reasoning, linear equations, and extended definitions. Lecture Notes in Computer Science, vol. 107. Springer-Verlag, New York, 1981, 1-43.
3. Bahlke, R., and Snelting G. The PSG - Programming System Generator. In Proceedings 1985 Conference on Language Issues in Programming Environments, (Seattle, 1985), 28-33.
4. Boyer, R., and Moore J. Proving theorems about LISP functions. J.ACM 22, 1 (Jan. 1975), 129-144.
5. Burstall ,R. M., and Darlington J. A transformation system for developing recursive programs. J. ACM 24, 1 (Jan. 1977), 44-67.
6. Dijkstra, E. W. A discipline of programming. Prentice-Hall, Englewood Cliffs, N.J., 1976.
7. Dybvig, R. K., and Smith, B. T. A semantic editor. In Proceedings 1985 Conference on Language Issues in Programming Environments, (Seattle, 1985), 74-82.
8. Gries, D. The science of programming. Springer-Verlag, New York, 1981.
9. Le Métayer, D. Mechanical analysis of program complexity. In Proceedings 1985 Conference on Language Issues in Programming Environments, (Seattle, 1985), 69-73.
10. Wadler, P. Applicative style programming, program transformation, and list operators. In Proceedings 1981 Conference on Functional Programming, Languages, and Computer Architecture, (Portsmouth, 1981), 25-32.
11. Williams, J. Notes on the FP style of functional programming. Advanced course on functional programming and its applications, (Newcastle upon Tyne, 1981)

Imprimé en France

par

l'Institut National de Recherche en Informatique et en Automatique