ABOUT PROGRAM TERMINATION AND SIMPLIFICATION

F. ERMINE (1) et I. GUESSARIAN (2)

ABSTRACT

We study two usual properties of program schemes: termination and a simplification property. To this end, we introduce the notion of useless occurrence which enables us to consider these two problems in a unifying setting; algebraic semantics provides us with a quite flexible framework for this study in that we can introduce the exactly desired amount of semantic information and derive the corresponding results.

We begin with a purely syntactic step, and, working in the free interpretation, obtain very easily at this step quite worthwhile simplifications. Then we introduce some amount of semantic information, and, working in a class of interpretations defined by algebraic conditions, show how we can delete some of the useless occurrences and non-terminating procedures with respect to that class of interpretations; this is achieved by using only the algebraic specifications of the program scheme and of the class of interpretations. Finally, we show how one can, making more and more precise the specifications of the interpretations, derive in the same way more and more refined simplifications of the program scheme.

(1) LCR, THOMSON-CSF, Domaine de Corbeville, 91401 ORSAY.

(2) CNRS - LITP, UER de Mathématiques de Paris 7, 2 Place Jussieu,
75221 PARIS CEDEX 05.

I - INTRODUCTION

It is now well-known that the most expensive and time consuming part of programming is debugging. Automatic transformations of high level programs [A, BD, H] has been shown recently to be a helpful step towards that goal. We study here such a transformation which will enable us to simplify programs and remove some useless loops. To this end we will use results from [G1] and extend them to general interpretation classes.

EXAMPLE I.1 : Consider the program block, clearly written with an absent mind, and which will loop :

```

j:=0;
LOOP :IF J#0 THEN GOTO EXIT
      IF A(K) < A(K-1) THEN exchange (A(K), A(K-1)) ;
      K:=K-1
      GOTO LOOP
EXIT : . . .

```

In [G1] we detected easily such a loop: to this end we generalized a code optimizing transformation [AU], consisting in the removal of useless variables and instructions; for particular values of the data, or the predicates, some instructions might never be executed, or might give rise to loops. Finding such loops is generally undecidable but it becomes decidable once we restrict our attention to program schemes instead of programs.

We thus begin with a purely syntactic stage and remove instructions - which for instance because of the nesting of recursive calls - will never be executed and point out a likely error.

EXAMPLE I.2 : Consider the famed recursive program :

$$F(x,y) = \text{IF } x = 0 \text{ THEN } 1 \text{ ELSE } F(x - 1, F(x - y, y))$$

which loops when called by value with $x=1$ and $y=0$. This is typically an example of a useless loop, since it comes from a branch of the program which never occurs in any terminating computation (namely the branch $F(x-y,y)$); removing that branch suppresses the associated loop.

The notion of useless occurrence, illustrated by example 1-2, is central to our simplification process. In [G1] we used it in a purely syntactic framework, and derived quite easily a very simple algorithm which performs nevertheless worthwhile simplifications. Applying now the same concept in a framework which contains some more semantic informations about the data (e.g. algebraic relations satisfied by the data and the base function symbols), we obtain somewhat more refined simplifications, such as the removal of the loop in example I-1. In this case however - in fact as soon as we do not deal with purely syntactic program schemes, and have some semantic knowledge - we can no more pretend to exhaustive results: i.e. although there is no hope of devising a general theory of loop detection (since this is un-

decidable), we are sketching here a partial theory enabling us to delete some of the loops in a program. We claim this is nevertheless worthwhile: e.g. our methods can help in obtaining safely better optimizing-compilers (by proving the correctness of the optimizations they will perform), or in improving debugging programs [BS].

From a slightly different point of view, related work has been done in [D] providing sufficient conditions for the termination of syntactic tree-rewriting systems, and is being extended to allow some semantic knowledge [KL]. Similarly, in studying λ -calculus models of programming languages [R], a way of modelling the relations holding between data and operations is to add δ -rules to the system: then, the termination and/or looping of the extended deduction system has to be studied anew. In [G1] such semantic knowledge was given by congruences (equivalences generated by equations between basic terms); however, most tests are not definable by congruences [G2]. So we introduce herein more general algebraic relations.

EXAMPLE I.3 : Assuming precise enough hypotheses on the interpretation domains, we can simplify by the previous methods the scheme [A] :

$$G(i,n)=\text{IF } i \neq n \text{ THEN } G(i+1,n) \text{ ELSE } h(n)$$

into the scheme: $G(i,n)=\text{IF } i \leq n \text{ THEN } h(n) \text{ ELSE undefined}$

we obtain here a more precise result than in [A] while assuming more general (i.e. less restrictive) hypotheses.

In this paper we study program schemes defined by recursive procedures or systems of mutually recursive equations: these are in fact the most general and the handiest ones in expressing programs and are particularly well suited to program transformations. For such program schemes we use the algebraic semantics framework [N] which is a very flexible tool in that it enables us to introduce at each stage the exactly needed amount of semantic information and derive the corresponding results; this in turn is quite helpful in assuming each time the minimal hypotheses needed for a result to be true: namely a strategy of minimal work for optimal results which should be the ultimate goal in Computer Science as well as in Mathematics !

Sacrificing rigor to readability, this paper contains very little formalism and is written at a very intuitive level. We first define our framework (§ II), then recall some results from [G1] about purely syntactical simplifications (§ III), while extending some of them to allow some semantic knowledge (§ IV); finally, we show we can add as much semantic information as needed and obtain correspondingly more and more accurate and refined simplification or termination results (§ V).

II - THE FRAMEWORK

Let $\{ f, g, h, \dots \}$ be a set of base function symbols, $\Phi = \{ F, G, H, \dots \}$ be a set of variable function symbols, $\{ \Omega, a_0, a_1, \dots \}$ be a set of constant symbols and $V = \{ v_1, v_2, \dots \}$ a set of variable symbols. Each function symbol φ has a rank $r(\varphi)$;

constant and variable symbols have rank 0, and Ω is a particular constant which corresponds to the "undefined". Let B denote the set of constant and base function symbols.

A term is:

- either an expression "e", e being a variable or constant symbol
- or an expression " $\varphi(\text{term}_1, \dots, \text{term}_r(\varphi))$ " φ being a function symbol.

A program scheme is a pair (S, u) where :

- S is a system of recursive equations: $F_i(v_1, \dots, v_{r(F_i)}) = u_i$
 $\forall i = 1, \dots, n$ u_i is a term containing only variables $v_1, \dots, v_{r(F_i)}$,
 $\Phi = \{F_1, \dots, F_n\}$ is the set of variable function symbols, representing the procedures of system S,
- u is a term, representing the "main procedure" of system S.

EXAMPLE II.1 : With the present notations, the program of example I-2 becomes the scheme (S, F(x,y)), where:

$$S : F(x,y) = g(x, a_0, a_1, F(p(x), F(d(x,y), y)))$$

To support intuition, we shall often use a tree-like representation of terms which gives here :

When one interprets constant and base function symbols, a program scheme becomes, via the choice of a semantics, namely an application associating with each scheme the function computed by that scheme, a program able to compute some function. For instance, interpreting g as an equality test, a_0 and a_1 as the constant 0 and 1, p as the predecessor function on integers and d as the substraction operation, we find back the recursive program of example I-2.

NOTATION: In the sequel we shall use a vector notation and abbreviate $G(v_1, \dots, v_{r(G)})$ (resp. $G(t_1, \dots, t_{r(G)})$) in $G(\vec{v})$ (resp. $G(\vec{t})$).

Algebraic semantics, relying on the notion of formal computation carried on by a scheme, is particularly well-suited to our purposes. We consider a scheme as a set of rewriting rules which, beginning with some initial term of the form u, macroexpands [DS] it by successive symbolic evaluations, thus defining a sequence of terms t_n which are the approximations to the computation of scheme (S, u). The function computed by the scheme is then obtained by interpreting the $t_n(\Omega)$'s - $t_n(\Omega)$ being the term obtained from t_n by substituting Ω 's to all subterms $G(\vec{t})$ of t_n . These semantics coincide with the fix-point one [N, S] as soon as the symbolic evaluations are done using the parallel call-by-name (or parallel OI) substitution rule [DS, N, V], we thus formally characterize the solution of scheme (S,u) by the

sequence of $t_n(\Omega)$'s, which can be described by an infinite tree $T(S, u)$.

EXAMPLE II.2 : (II.1 continued): The first terms t_n associated with scheme $(S, F(\vec{x}))$ are the following :

They are associated with the following sequence of $t_n(\Omega)$'s :

which in turn is associated with the following infinite tree $T(S, F(\vec{x}))$:

The reader will easily be convinced that, when interpreted as in example II-1, $T(S, F(\vec{x}))$ provides us with the function computed by the program of example I-2.

III - ABOUT PURE SYNTAX

We can now express the looping of a program in this framework: a function variable F is said to loop or to be non-terminating iff $T(S, F(\vec{V})) = \Omega$. In this case, and for any interpretation, the program deduced from S with main procedure F will loop. The first step in any simplification process will thus be the removal from scheme S of function variables F which loop, as well as equations defining such F 's. This can be done quite easily and a very simple algorithm for doing so is given in [G1]. From now on, our schemes will be supposed to have no looping function variable. With example I.2 in mind we now introduce the main definition of this section.

DEFINITION III - 1 : An occurrence of a term t is useless in program scheme (S, u) iff it can be replaced by Ω without changing $T(S, u)$.

EXAMPLE III - 2 : The occurrence of $F(d(x, y), y)$ is useless in the right-hand side of scheme S in example II-1.

Remark : The notion of useless occurrence is different from the one of hopeless node of [DS]. The latter is relative to one fixed term whereas the former takes into account all the approximations of the computation of scheme (S, u) . For instance p and the outermost occurrence of F are hopeless in t_2 (Example II - 2), but useful, because they will be used later to compute $t_3(\Omega)$. However, a function variable F is said to be hopeless iff F loops, equivalently iff F is useless in $(S, F(\vec{V}))$.

One can prove [G1] that an occurrence may be useless because of exactly two reasons:

1) either it occurs as an argument to a looping function variable.

2) or, as in example II-1, it occurs in a useless argument of a function variable namely an argument v_i of an F which is useless in $(S, F(\vec{v}))$.

The first case is excluded since our schemes have no looping function variables; we give below an algorithm which deletes all useless arguments and thus enables us to construct from any given scheme a reduced one having no useless occurrence. This algorithm relies on the intuitively clear fact that an argument v_i is useless iff v_i does not appear in $T(S, F(\vec{v}))$.

ALGORITHM III - 3 : 1) Find out the useless arguments in the equations of S

1.1) Construct a set $B = \{ b_{ij}, j=1, \dots, r(G_i), i=1, \dots, n \}$ of boolean variables and the system $B(S)$ of boolean implications: $\text{true} \rightarrow b_{ij}$ if there exists an occurrence of v_j in u_i all the ancestors of which are labelled by base function symbols.
 $b_{i1} \wedge b_{i2} \wedge \dots \wedge b_{ik} \rightarrow b_{ij}$ for every occurrence of v_j in u_i which has k ancestors labelled by elements $G_{i_1}, \dots, G_{i_k}$ of Φ and which occurs in the j_p^{th} argument of G_{i_p} , for $p=1, \dots, k$.

The occurrence of argument v_j in the left hand side of equation defining G_i is useful if and only if $B(S) \vdash (\text{true} \rightarrow b_{ij})$.

1.2) Construct $B_{\text{true}} = \{ b_{ij} / B(S) \vdash (\text{true} \rightarrow b_{ij}) \}$. It is defined inductively

by: $B_0 = \{ b / (\text{true} \rightarrow b) \in B(S) \}$

$B_{i+1} = B_i \cup \{ b / \exists b_1, \dots, b_k \in B_i, (b_1 \wedge \dots \wedge b_k \rightarrow b) \in B(S) \}$

$B_{\text{true}} = \text{lub } B_i$

1.3) Let $B_{\text{false}} = B - B_{\text{true}}$. v_j is useless in $G_i(v_1, \dots, v_{r(G_i)})$ if and only if

$b_{ij} \in B_{\text{false}}$.

2) Reduce S

2.1) Suppress the useless arguments v_j in the left hand sides $G_i(v_1, \dots, v_{r(G_i)})$

2.2) Replace in the u_k 's each $G_i(m_1, \dots, m_{j-1}, m_j, m_{j+1}, \dots, m_{r(G_i)})$ by $G_i(m_1, \dots, m_{j-1}, m_{j+1}, \dots, m_{r(G_i)})$ whenever $b_{ij} \in B_{\text{false}}$

2.3) Replace each G in Φ by a new symbol G' .

Intuitively, b_{ij} represents the truth-value of the predicate " v_j is useful in $(S, G_i(\vec{v}))$ ". A formal justification of this algorithm and of the subsequent theorem can be found in [G1] :

THEOREM.III - 4 : For any scheme (S, u) the previous algorithm provides us with a scheme (S', u') , having no useless occurrence, and such that $T(S, u) = T(S', u')$, i.e. computing the same function as (S, u) for every interpretation.

EXAMPLE III - 5 : Applying the previous algorithm to the scheme of Example II-1, we simplify it into $(S', F'(x))$ defined by :

$$S' : F'(x) = g(x, a_0, a_1, F'(p(x)))$$

and clearly satisfying $T(S', F'(x)) = T(S, F(\vec{x}))$

IV - ABOUT SOME SEMANTICS

In the previous section, we removed all variables and terms which were looping or useless because of purely syntactical reasons. However, even in such reduced schemes, loops or useless occurrences may surge, due to semantics.

EXAMPLE IV - 1 : Consider the scheme $(S, G(x))$:

$$S : G(x) = g(x, g(x, G(h(x)), G^2(k(x))), k(x))$$

it is reduced by the standards of section 3. Suppose now g is always interpreted as a conditional (if ... then ... else ...) then the branch $G^2(k(x))$ is useless and should be deleted. With the assumption that g is a conditional $(S, G(x))$ should be reduced into : $S' : G(x) = g(x, G(h(x)), k(x))$.

In this section, we shall concentrate on such semantical simplifications. We first introduce a few more definitions about the framework.

Let C be a class of interpretations - e.g. the interpretations I such that g_I is a conditional. Two program schemes are C-equivalent - notation $\equiv_C$ - iff they compute the same function for every interpretation I in C . One has the following equivalences :

$$(S, u) \equiv_C (S', u') \Leftrightarrow T(S, u) \equiv_C T(S', u') \Leftrightarrow \forall I \in C \ T(S, u)_I = T(S', u')_I$$

EXAMPLE IV - 1 (continued) : Let $\equiv$ be the congruence (a congruence is here an equivalence compatible with the algebraic structure and substitution closed) generated by the axiom (1) $g(x, g(x, y, z), w) = g(x, y, w)$ and C the class of interpretations I compatible with (1) - i.e. such that $t \equiv t' \Rightarrow t_I = t'_I$.

C clearly contains the interpretations where g is interpreted as a conditional. Then $(S, G(x)) \equiv_C (S', G(x))$, and the simplification from S to S' is trivially carried on applying axiom (1) only to the right-hand side of S .

Let $L(S, u) = \{ t / t < T(S, u), t \in M(B, V) \}$ be the set of finite approximations to $T(S, u)$. Then a function variable G is said to loop with respect to C iff $T(S, u) \neq_C \Omega$ or equivalently, $\forall t \in L(S, u) \ t \equiv_C \Omega$. Similarly, an occurrence of a term is said to be useless w.r.t.C iff it can be replaced by Ω without changing :

$$L_C(S, u) = \{ t / \exists t' \in L(S, u) \ t \equiv_C t' \}$$

In the sequel, our simplifications will rely on the following lemma, which gives an easy sufficient condition of C -equivalence between two schemes.

LEMMA IV - 2 : Let (S, u) , (S', u') be two schemes and $t_n(\Omega)$, $t'_n(\Omega)$ the sequences associated with them in section II. If for each n there exists an $m(n)$ such that $t'_n(\Omega) \equiv_C t_{m(n)}(\Omega)$, and $\lim_{n \rightarrow \infty} m(n) = \infty$, then $(S, u) \equiv_C (S', u')$.

In this section, C will be a congruence, generated by a set of equalities between finite terms as in Example IV-1, C the class of interpretations compatible with C - or satisfying the equalities defining C - and we shall concentrate on removing loops and useless occurrences w.r.t. C , e.g. as in Example IV-1. Of course we can no more hope to obtain here a general algorithm removing all useless occurrences since this problem contains the word problem and hence is unsolvable. But we will get handy sufficient conditions for simplifications to take place.

We shall look first at termination properties - namely looping function variables. The first step is of course as in Example IV-1.

ALGORITHM IV - 4 : Simplification 1 : Apply the rules of C to the right-hand sides of the equations defining S to simplify them as much as possible.

The second step relies on a very simple idea and does however surprisingly well in a number of cases.

Simplification 2 : Look whether $T(S, G(\vec{v})) = \Omega$ is a solution to scheme S modulo the rules of C . In this case, and since we are looking for the least solution of S in any interpretation and nothing can be smaller than Ω , we know that G loops.

EXAMPLE IV - 5 : Let C be defined by the equality : (2) $g(z, u, v) = u$ which is satisfied in particular whenever g is interpreted as a conditional testing whether its first argument is null. Now the program of example I-1 is associated with the scheme $(S, G(v))$:

$$\begin{aligned} S : \quad & G(v) = H(z, v) \\ & H(i, v) = g(i, H(i, b(v, c(v))), K(i, v)) \\ & K(i, v) = \dots \end{aligned}$$

and all interpretations satisfy (2). Then $T(S, G(v)) = \Omega$ is clearly solution of S since we obtain, replacing $G(v)$ by Ω in S : $H(z, v) = \Omega$
but $H(z, v) = g(z, H(z, b(v, c(v))), K(z, v)) = h(z, b(v, c(v)))$ by (2)
hence $H(z, v) = \Omega$ satisfies the second equation of S .

EXAMPLE IV - 6 : Let S be the program :

$$G(v) = \text{if } v = 0 \text{ then } G(v + 1) \text{ else } G(v - 1)$$

it is associated with the scheme $S : G(v) = g(v, G(s(v)), G(p(v)))$

and the interpretations we consider satisfy axiom (3) $g(v, \Omega, \Omega) = \Omega$.

Then one trivially verifies that $T(S, G(v)) = \Omega$ is a solution of scheme $(S, G(v))$

w.r.t. the interpretations satisfying (3).

In some cases however, due to its special status, Ω is excluded from the rules of C; then a more operational way of recognizing looping function variables proceeds as follows :

We suppose we can choose for each term t a term t' , C-equivalent to t ; possibly - if C is Church Rosser for instance - the minimal such t' should be chosen.

ALGORITHM IV - 7 : 1) Macroexpand $G(\vec{v})$ according to S and simplify the result of the macroexpansion using the rules of C; beginning each time with the previously obtained term, repeat these two operations n times, where n is the number of equations of S. Let $G(\vec{v}) \xrightarrow{S} t_1 \xrightarrow{C} t'_1 \xrightarrow{S} t_2 \dots \xrightarrow{S} t_{n+1} \xrightarrow{C} t'_{n+1}$ be the sequence thus obtained.

2) If two among the t'_j 's, say t'_i and t'_{i+k} have a root which is the same function variable, e.g. G_i , which gives $t'_i = G_i(\vec{t}(\vec{v}))$ and $t'_{i+k} = G_i(\vec{t}'(\vec{v}))$, if moreover t'_{i+k} is an instance of t'_i , namely $\vec{t}'(\vec{v}) = \vec{t}(\vec{t}'(\vec{v}))$, and if the constant symbols which occur in $\vec{t}$ and are actually used in simplifications going from t'_i to t'_{i+k} , are not modified then the sequence of macroexpansions and simplifications from t'_i to t'_{i+k} can be repeated forever and G loops.

3) Otherwise no conclusion is possible.

EXAMPLE IV - 8 (IV - 5 continued): Applied to this Example, the previous algorithm gives the sequence:

$G(v) \xrightarrow{S} t_1 = t'_1 = H(z, v) \xrightarrow{S} t_2 = g(z, H(z, b(v, c(v))), K(z, v)) \xrightarrow{C} t'_2 = H(z, b(v, c(v)))$
whereby matching t'_2 to t'_1 we can conclude that G loops.

The previous algorithm is based essentially on a kind of symbolic evaluation. The same idea can be applied looking not only at termination properties or looping function variables, but also at simplifications which consist in removing useless occurrences or variables. An algorithm can be given, based on the ideas of algorithms III-3 and IV-7 to remove some of the useless variables [G1]. However, here, the removal of useless variables is no more sufficient to ensure the removal of useless occurrences as shown by the following example :

EXAMPLE IV - 9 : Let (S, $G(v)$) be defined by :

$$\begin{aligned} S : G(v) &= g(v, H(v), G(h(v))) \\ &: H(v) = g(v, G(v), k(v)) \end{aligned}$$

and suppose all the interpretations we consider satisfy axiom (1) - which is clearly the case if g is interpreted as a conditional. Then the only variable v is clearly useful, nevertheless (S, $G(v)$) can be simplified in

$$S' : G(v) = g(v, \Omega, G(h(v)))$$

The method to recognize such useless occurrences proceeds along lines similar to algorithm IV-7.

ALGORITHM IV - 10 : Let H be an occurrence of a function variable in a term which we single out by writing that term as $t(\vec{u}, H(\vec{u}'))$ where $\vec{u}$ and $\vec{u}'$ are vectors of terms; suppose moreover no function variable occurs in t .

1) Let $t_0 = t(\vec{u}, H(\vec{u}'))$ and $t_0 \xrightarrow{S} t_1 \xrightarrow{C} t'_1 \xrightarrow{S} t_2 \xrightarrow{C} t'_2 \dots$ be the sequence obtained by successively macroexpanding and simplifying t_0 as in algorithm IV-7.

2) If t is left unchanged - or occurs periodically - in that process and function variables always occur at the same place as H , then if one of the t'_j 's has the form: $t'_j = t(\vec{u}_j, H(\vec{u}'_j))$ and if the macroexpansions and simplifications applied between t_0 and t'_j can be repeated identically beginning with t'_j - and this happens if the constants which are actually used in simplification rules are left unchanged as well as similar terms used in non-linear simplification rules (e.g. the x of rule $g(u, x, h(x)) = x$) - then H is useless in t_0 and t_0 is C-equivalent to $t(\vec{u}, \Omega)$.

Of course this method implies a choice - nothing but arbitrary - of when to stop the sequence of t'_j 's.

EXAMPLE IV - 11 (IV - 9 continued) : Beginning with $t_0 = g(v, H(v), G(h(v)))$ we obtain here the sequence: $t'_1 = g(v, G(v), g(h(v), H(h(v)), G(h^2(v))))$ whereby we can conclude that the equation defining H is useless and can be deleted,

$t'_2 = g(v, H(v), g(h(v)), \dots)$ whereby we can conclude that H can be replaced by Ω in S , whence S' .

However the previous methods rely mainly on syntax and many usual - and trivial - loops are of a more semantical nature, for instance the following one, which could not be detected by these methods :

EXAMPLE IV - 12 : LOOP : IF $I \geq 0$ THEN $I := I+1$
GOTO LOOP

It is associated with the scheme: $S: G(i) = g(i, a_0, G(s(i)), G(i))$; g is a 4-ary basic function symbol which will be interpreted as the conditional "greater than" i.e.: $g_I(u, v, w, t) = \text{IF } u \geq v \text{ THEN } w \text{ ELSE } t$; a_0 is a constant symbol, and $a_{0I} = 0$ is the zero constant; s is a basic function symbol which will be interpreted as the "successor" function on the integers, i.e.: $s_I(x) = x+1$.

The next section is mainly devoted to recognizing such loops and useless occurrences related to order theoretical conditions.

V - MORE ABOUT SEMANTICS

In this section, the interpretation domains D_I are supposed to be ordered by a relation $\leq$: for instance in PASCAL (or PL1), basic sets are ordered, e.g.:

$1 \leq 2 \leq \dots \leq 9 \leq A \leq B \leq \dots \leq Z$. The 4-ary base function symbol g will always be interpreted as in example IV-12 (the other conditionals "less than", "strictly less than" etc ... can be treated similarly).

DEFINITION V - 1 : Let f_I be a n -ary function over D_I ; f_I is said to augment (resp. decrease) w.r.t. argument v_i iff $\forall v_1 \dots v_n$
 $f_I(v_1, \dots, v_i, \dots, v_n) \geq v_i$ (resp. $f_I(v_1, \dots, v_i, \dots, v_n) \leq v_i$)
 the notation is : f_I has property A_i (resp. D_i).

Remarks: 1) These properties may depend upon the actual values of the other arguments of f_I . The looping on a conditionnal g will depend upon the initial value of the test variable and the properties A or D of the functions subsequently applied to it.

2) It is not possible here to use a congruence since we know [G2] that tests are not definable by congruences.

To shorten, we suppose here there are no other simplification rules than hypothesis (4) and its consequences - e.g. rule (1); the case where the interpretations satisfy some additional rules - e.g. congruences - can be treated similarly but extra care has to be paid for rules involving constants and non-linear rules (see the previous sections).

The following algorithm implies some knowledge about the properties A_i and D_i of the interpretations of base function symbols.

ALGORITHM V - 2 : 1) - Suppose - this is possible without loss of generality - that we have in system S , an equation : (5) $G(\vec{V}) = g(v_i, a_n, u, u')$
 2) - If, after "some" macroexpansions and simplifications u is rewritten into $G(u_1, \dots, u_n)$, if moreover u_i is a term having property A_i w.r.t. argument v_i , then G will loop when beginning with an argument $v_{iI} \geq a_{nI}$ and the equation defining G can be replaced by :

$$G(\vec{V}) = g(v_i, a_n, \Omega, u')$$

3) Similarly, if u' is rewritten in $G(u'_1, \dots, u'_n)$ and u'_i has property D_i w.r.t. v_i , then equation (5) becomes:

$$G(\vec{V}) = g(v_i, a_n, u, \Omega)$$

We apply here this algorithm to two typical examples.

EXAMPLE V - 3 (IV - 12 continued) : $\leq$ is here the usual order on the integers which does not coincide with the canonical order $\leq_I$ ($\leq_I$ is the discrete order where all elements are incomparable, but for the least element ω). By the method of section IV.1, this scheme can be reduced into: $S': G(i) = g(i, a_0, G(s(i)), \Omega)$; now, knowing that s augments its only arguments i - this knowledge is given together with the interpretations we consider - algorithm V.2 enables us to

simplify S' in: $S'' : G(i) = g(i, a_0, \Omega, \Omega)$; whence we deduce again by section IV : $S'' : G(i) = \Omega$

Our methods apply equally easily to program simplifications as to program termination.

EXAMPLE V - 4 : Consider the program of Example 1 - 3, which is obtained in [A] as the result of some program transformations. In our formalism, it becomes :

$$S : G(i, n) = g(n, i, g(i, n, h(n), G(i+1, n)), G(i+1, n))$$

concentrating first on the inner g , and knowing that the domains D_I of our interpretations are archimedian, we can conclude by a method similar to algorithm V - 2, (but using the archimedian property instead of just the order structure) that: $g(i, n, h(n), G(i+1, n))$ can be replaced by $h(n)$; then S becomes : $S' : G(i, n) = g(n, i, h(n), G(i+1, n))$; and by algorithm V - 2 : $S'' : G(i, n) = g(n, i, h(n), \Omega)$, which is the form announced in Example 1 - 3.

We concentrated here on the order structure of the domains of interpretations and on conditionals based on order tests. Of course other types of conditionals would lead to similar results, while singling out different properties of the interpretations. For instance if :

$$S : G(i) = g(i, G(i \text{ or } \text{TRUE}), G(i))$$

$$\begin{aligned} \text{with } g_I(x, u, v) &= u \text{ if } x = \text{TRUE} \\ &= v \text{ if } x = \text{FALSE} \\ &= \text{undefined otherwise} \end{aligned}$$

and if we know that $x \text{ or } \text{TRUE}$ is always true for any boolean x , then S can be reduced to $S' : G = \Omega$.

Notice finally that there is of course some arbitrariness in algorithm V - 2: namely the "some" in step 2, i.e. the choice of where to stop the macroexpansions. And this, especially in the case where we allow additional simplifications rules, ensures us that we shall never be able to detect all loops: for if we decide to stop after N macroexpansions, one can always devise a program in which a loop will appear after $N + 1$ steps. We believe however that our methods are quite helpful tools in deleting most of the usual loops, which are the effect of inadvertance, or useless terms which may appear in results obtained by automatic program transformations [A, H, K].

Acknowledgments: We are glad to thank J.F. PERROT for his careful reading and judicious comments, C. HENRY, all members of the LCR- THOMSON and especially its director G. RUGGIU, and the referees for helpful suggestions, and Mrs SAKOUN for her patient typing.

REFERENCES

- [A] J. ARSAC,
"La Construction de programmes structurés"
ed. DUNOD, PARIS (1977)
- [AU] A.V. AHO, J.D. ULLMANN,
"The theory of parsing, translation and compiling"
Vol. 2, Prentice Hall, LONDON (1973)
- [BD] R.M. BURSTALL, J. DARLINGTON,
"A transformation system for developing recursive programs"
J.A.C.M. 24, p.44-67 (1977)
- [BS] J.P. BENSON, S.H. SAIB,
"A software quality assurance experiment"
ACM Software Engineering Notes, Vol. 3, N° 5, p. 87-91 (November 78)
- [D] N. DERSHOWITZ,
"Orderings for term rewriting systems"
FOCS Conf., Porto-Rico (1979), p.123-131.
- [DS] P.J. DOWNEY, R. SETHI,
"Correct computation rules for recursive languages"
SIAM J. COMP. 5, p. 378-401 (1976)
- [G1] I. GUESSARIAN,
"Program transformations and algebraic semantics"
T.C.S. 9, p. 39-65 (1979)
- [G2] I. GUESSARIAN,
"Les tests et leur caractérisation syntaxique"
RAIRO 11, p. 133-156 (1977)
- [H] C. HENRY,
"Résolution d'équations algébriques sur le magma libre - Application
aux transformations de programmes"
Thèse de 3ème cycle - Université de PARIS VII (1978)
- [K] L. KOTT,
"About a transformation system : a theoretical study"
in "Program transformations", 3rd ISP, Ed. DUNOD, p. 232-247 - PARIS (1978)
- [KL] S. KAMIN, J.J. LEVY,
"Attempts for generalizing recursive path orderings"
in preparation
- [N] M. NIVAT,
"On the interpretation of recursive polyadic program schemes"
Symp. Math. 15, p. 256-281 - ROME (1975)
- [P] J.F. PERROT,
"Programmes, mots et arbres"
Rend. sem. Mat. Univ. Politecn. Torino 36, p. 11-14 (1977-1978)
- [R] B. ROBINET,
"Types et fonctionnalité",
in "Lambda-Calcul et Sémantique formelle des langages de programmation"
6e Ecole de Printemps (La Châtre 1978), p. 303-325.
- [S] D. SCOTT
"Data types as lattices"
SIAM J. COMP. 5, p. 522-587 (1976)