SOME ASPECTS OF THE STATIC SEMANTICS OF LOGIC PROGRAMS WITH MONADIC FUNCTIONS

Patrizia Asirelli

Ist. di Elab. dell'Informazione - C.N.R.
V. S. Maria, 46 - I56100 PISA - Italy

ABSTRACT

We consider logic programs in the Horn clauses form of logic with monadic functions, and present two approaches to derive a set of equations from a given set of clauses. The derivation is obtained by a data flow analysis of the variables, involved in each clausal definition. Each equation expresses the semantics of a procedure, by means of a set expression which, by transformation of the set of equations, can be reduced to a solved form. The set expression thus obtained represents, for each procedure its greatest, approximate, set of solutions; i.e. a set which contains the denotation defined, for the same procedure, by the standard semantics. The approximate solutions can be seen in the contest of abstract interpretations of programs, to get, by a static analysis of their definitions, some of their properties. The approximate solutions, being expressed by means of set expressions, could then be used as a tool for program verification and construction.

1. INTRODUCTION

We present an approach to the static analysis of programs, written in a simple logic language, defined as in |1-2|, where procedures are defined by means of Horn clauses with monadic functions, and where all clauses are non negative. Thus a sort of very simple PROLOG |3-5|.

We first define an algebraic semantics of clausal definitions, in the sense of representing possible set of solutions for a procedure, by means of equations and set expressions.

By static semantics of a program we mean all that can de deduced, statically, about the set of solutions for a logic program, as expressed by its standard semantics |2|.

The aims of this paper fall into the same framework of |6|, but for logic languages instead of algorithmic languages. As in |6|, we get set expressions which represent in the most general case, approximation to the set of denotations of a procedure as defined by the standard semantics. We also consider And/Or graphs |7-8|, representation of programs instead of flow-charts. Thus, differently from |9|, we do not try to get set expressions which denote the exact set of solutions, but only an approximation of it. At present the work is much semplified, with respect to |6| and |9|, since we consider problems originated only by the use of monadic functions, treated symbolically, without looking, for now, for the fixpoints of their associated symbolic expression.

The same problem as been tackled for monadic logic programs with monadic functions; the next Section will give a brief summary of results obtained, for that case, in |10|. Section 3 will present two approaches to deduce, statically, a set of equations from a given set of clauses. In Section 4 we will present transfromations of such equations to get a set of equations in solved form. Section 5 contains few considerations on the defined transformations and their relations to other works. Section 6 extends the results of the previous sections, to clausal definition with monadic functions. We conclude with a brief summary in Section 7.

Appendix 1, at the end of the paper, gives an example of the construction of a set of equations for a given set of clauses, in the monadic case; Appendix 2 gives an example of a set of equations that can be obtained, according to the second approach presented in Section 3. In Appendix 3 a set of axioms is given to be used for transformations of equations obtained when functions are used in clausal definitions.

2. RESULTS FROM THE MONADIC CASE

Given a set of clauses A, defining n procedures Pi, monadic, we consider the set of clauses defining each procedure and its correspondent And/Or graph,|7-8|. Then we trace the values flow of the variable appearing at the root of the And/Or graph. The denotation of a procedure Pi, Dh(Pi), (results of the procedure Pi), can be derived in terms of union and intersection of denotations of the predicates involved in the definition of Pi. Thus we can derive, say, an $\cap/\cup$ graph, correspondent to the And/Or graph in object, by interpreting And nodes and Or nodes, respectively as intersection and union operations, and replacing each atomic formulas Qj(X) (where X is the variable traced), by the correspondent set Dh(Qj). By Dh(Pi) we denote the denotation of the predicate Pi, as defined by the operational semantics associated to Hyperresolution and Instantiation rules, |2|. Appendix 1 shows an example of the all process; from the $\cap/\cup$ graph there obtained, the following equation can be derived:

$$\{Pi\} == \left(\bigcap_{j=1}^{n} \{Qj\}\right) \cup \left\{f.\left(\bigcap_{j=1}^{m} \{Rj\}\right)\right\} \cup \left(\bigcap_{j=1}^{k} \underline{\mathrm{dom}}(gj,\{Sj\})\right) \cup \{a\} \cup \{b\}$$

Where:
each $\{Rj\}$, $\{Qj\}$ denotes the set of values, solutions of procedures Rj and Qj, respectively, derived from the static analysis of their clausal definition.
$\{f.T\}$, is a notation which, given a set (or set expression) T, denotes a set of data that, using a common set notation, can be expressed as: $\{ f(y) \mid y \in T \}$.
dom(gj, {Rj}) denotes:- a set expression T such that: $\{gj.T\} \in \{Rj\}$;
- the empty set, {}, otherwise.

We have called '__Deduce__' the function which produces a set of equation such as the above one, starting from a given set of clauses. Then we show, inductively, that when predicates are defined by clauses where: functions symbols are not used or else, they are used not recursively, then the set expression denoted by {P} can be computed to a set of values such that the following relation holds:

$$t \in Dh(P) \text{ iff } t \in \{P\} \quad \text{i.e. } Dh(P) \equiv \{P\}$$

When clauses use function symbols recursively, either directly or not, that is, when clauses are such as follows:

P(f(X)) <- Q(X), P(X)

or

P(X) <- Q(X), R(X)
Q(f(X)) <-P(X)

then we __Deduce__ equations of the form:

{P}== {f.T}

where T contains references to the symbolic set {P} itself. In those cases we need to find the fixpoint of such set expressions T. Then, if we are able to find a notation for such fixpoints so that by replacement of T in {f.T} we get a notation which is recursive, but self-contained and, if we are then able to define union and intersection between such sets then we can represent the denotation of a predicate by a set expression which is finite, independent of other predicates and which can be built by the static analysis of clauses. In |10| we suggest such a notation and give tranformation rules for deducing such notations from the set of equations obtained in the first place. On the other hand we show that those set expressions can be left symbolic, and transformed ({f.T}'s are only partially transformed), to obtain a set of equations, in solved form, which represent a new set of clauses. The new set of clauses are the semplified version of the set of clauses given in the first place. Thus, we defined an algorithm Transf such that the following diagram commutes:

$$\begin{tikzcd}
& A \arrow[r, "\underline{Deduce}"] \arrow[dd, "Transf"'] \arrow[dl] & EA \arrow[dd, "Transf"] \\
DH & & \\
& A' \arrow[r, "\underline{Deduce}"'] \arrow[ul] & EA'
\end{tikzcd}$$

fig. 1

Where DH={Dh(P1),Dh(P2),....Dh(Pn)}, for all predicates P1,..Pn defined in A and A'.

That is, the set of equations EA' can also be derived from a set of clauses A', such that: denoting by $Dh_A(Pi)$, the denotation of Pi as defined by the standard semantics, when Pi is defined in a set of clauses A; then: A' and A are such that, for all procedures Pi defined in A, Pi is defined also in A' and:

$Dh_A(Pi) \equiv Dh_{A'}(Pi) \equiv Dh(Pi)$ if clauses in A do not contain local variables;

$Dh_A(Pi) \subseteq Dh_{A'}(Pi)$ otherwise.

Moreover, A' can be obtained by transformations of A.

Equations EA', obtained by transforming equations EA, may contain references to symbolic sets {Pi}, where Pi is an undefined predicate. Such symbolic sets can be eliminated (replaced by the empty set {}). In any case such equations either represent a ground set of values or else, they may be considered as patterns for the computation of them. They can also be used as a tool for programs development and programs composition, where symbolic set are used to define parametric specifications.

3. TWO APPROACHES TO THE EXTENTION OF DEDUCE

To introduce the problem of static semantics, for n-adic programs, let's consider the following clauses:

1) P(a,b)<-
2) P(a,X)<-
3) P(X,Y)<-Q(X,Y),P(X,Y)

4) P(X,Y)<- Q(X,Z) P(Z,Y)

In general we have clauses such as:

P(X1, X2,...Xn)<- Q1(t11,..tm1),...Qn(tn1,....tnk)

For the moment we do not consider functions, thus we assume all terms tij to be variables, either local or not.

We denote by {P} a set of tuples, each one of which is meant to represent a solution for P, according with the definition of Dh(P) in the same case.

Let's observe that Dh(P) in case of clause 1) is given by {<a,b>}; thus an obvious way to modify the <u>Deduce</u> function , is to produce the same results for clauses which are assertions. On the other hand, following the same approach, for the second clause we get a tuple such as {<a,?>}, where ?, means "all possible values", [10].
For the same clause it is:

$$Dh(P)=\{ <a,\ t> \mid \forall\ t \in \text{Herbrand Universe}\}$$

if we let {?} denote the Herbrand Universe of the set of clauses defining P, we can represent Dh(P) as: {a} X {?}. An obvious way to make things equal is then to define the cartesian products between sets as usual, so that:
{<a,?>}=={a} X {?}
as if ? were an other constant symbol. The semantics of {<a,?>} has to be defined as Dh(P) above, so that {P} and Dh(P) represent the same set of values. In general, a tuple as <a,b,?,c,d> represents a set of tuples whose first two and last two projections are fixed, and the middle one is one of all possible data. Given the meaning of such a notation, we can redefine <u>Deduce</u> so that, for all assertions, the following set expression will be constructed:

$$\{< v_1,\ v_2,\ \ v_n >\} \quad \underline{\text{iff}} \quad C : P(t_1, t_2, \ldots\ldots, t_n) <-$$

$$\text{for all } v_i = t_i \text{ if } t_i \text{ is a constant symbol}$$

$$v_i = \text{'?' if } t_i \text{ is a variable.}$$

Let P be a m-adic procedure and let it be defined by n assertions; then we can deduce the following equation:

$$\{P\} == \{ < v_1^1,\ v_2^1, \ldots\ldots v_m^1 >, <v_1^2,\ v_2^2, \ldots v_m^2>, \ldots\ldots <v_1^n,\ v_2^n, \ldots v_m^n>\}$$

If a procedure P has m arguments, than given the set of tuples {P},we define the m projections of {P}, each one denoted by {P}_j, j=1:m. Thus, each {P}_j denote the set of values for the j-th argument of predicate P, and it is defined by:

$\{P\}\_j == \{ v^i_j \mid \forall\ i=1{:}n \text{ and } \forall < v^i_1, ..v^i_j, ...v^i_m> \in \{P\} \}$

Given {P} as above, it is, obviously:

$$\{P\} \subseteq \prod_{j=1}^{n} \{P\}\_j$$

The same holds when we Deduce each {P}_j, by the data flow analysis of variables, for all other type of clauses. We present two approaches: one leads to a set of equations which allows to find, for each given procedure Pi, an approximation of Dh(Pi); the other approach leads to a set of equations which could be refined to find an approximation of Dh(Pi), which is the closest one that can be found statically, by the data flow analysis of variables.

3.1 First approach

Like for the monadic case, we consider the And/Or graph which correspond to a clause; then we trace the value flows of all variables, arguments of the predicate being defined. For each variable Xi such that its trace binds it to the j-th argument of a call to procedure Q, we consider {Q}_j as the set originating values for that variable. As in the monadic case we then interpret as intersection all And nodes and as union all Or nodes. Just as an example, let us see that, proceeding as above, for clause 3 we would deduce:

{P}_1== {Q}_1 ∩ {F}_1
{P}_2== {Q}_2 ∩ {F}_2

While for clauses 4:

{P}_1== {Q}_1
{P}_2== {F}_2

We can see that {P}_1 X {P}_2 derived above, do not represent correctly the semantics of P as defined by the standard semantics for the corresponding clauses. In general, while for assertions, {P} and Dh(P), represent the same set of data, for all other clauses we have:

$Dh(P) \subseteq \{P\}$

That is, the static semantics, defined by the data flow analysis of variables, defines for a predicate P, a denotation which contains the denotation defined by the standard semantics. We can easily see that, for example, the standard semantics defines for P, relatively to clause 4, the following:

$Dh(P) == \{ <t1,t2> \mid \forall <t1,k> \in Dh(Q) \text{ and } <k,t2> \in Dh(F) \}$

while the set expression we get for {P}, can be expressed as:

$\{P\} == \{ \langle t_1, t_2 \rangle \mid \forall\ k_1, k_2 : \langle t_1, k_1 \rangle \in \{Q\}$ <u>and</u> $\langle k_2, t_2 \rangle \in \{F\} \}$

To conclude, given a set of clauses A, for each predicate Pi, defined in A, by this approach we would get a set of equations as follows:

$$\{Pi\} == \bigcup_{j=1}^{n} \prod_{k=1}^{m} \{Pi\}j\text{-}k$$ for Pi defined by n clauses, and Pi being a m-adic procedure.

$\{Pi\}j\text{-}k == T$ for T a set expression defining the set of possible values for the k-th argument of Pi, defined by the j-th clause.

Let's observe that this way we find the greatest approximation of each Dh(Pi) that can be found by this method. Let's also observe that the set of equations EA are such that the following diagram commutes:

$$\begin{tikzcd} DH(A) \arrow[d] & A \arrow[l] \arrow[r, "\underline{Deduce}"] \arrow[d, "Transf"'] & EA \\ DH(A') & A' \arrow[l] \arrow[ur] & \end{tikzcd}$$

fig. 2

Where $DH(A) = \{Dh_A(P1), Dh_A(P2), \ldots Dh_A(Pn)\}$

$DH(A') = \{Dh_{A'}(P1), Dh_{A'}(P2), \ldots Dh_{A'}(Pn)\}$, and $Dh_A(Pi)$, $Dh_{A'}(Pi)$ defined in Section 2, fig. 1.

In fact we can easily prove that, given a clause with local variables that binds together some procedure calls, ignoring local variables completely, we get an approximation of its denotation, which is the denotation of an analogous clause, where all local variables, in all procedure calls are different, one from each other. That is, given, for example:

P(X,Y) <- Q(X,Z1), F(Z1,Y), R(Y,Z2), M(Z2,Z1)

Indeed we find the denotation of P defined as:

P(X,Y)<- Q(X,Z1), F(Z2,Y), E(Y,Z3), E(Z4,Z5)

Thus we get a set of equations EA, that can also be derived from a set of clauses A', such that the denotation of all procedures, defined in A are contained in the denotation of the same procedures defined in A'. I. e.
$Dh_A(Pi) \subseteq Dh_{A'}(Pi)$ for all procedures Pi.
The previous relation shows that by ignoring local variables, the method of data flow analysis of variables, ensure partial consistency with standard semantics. In fact, the method allows to find sets of values such that, some of the values are correct solutions, while others aren't. Yet no value, outside those sets, can be a correct solution.

3.2 <u>Second approach</u>

To obtain a set expression for P representing a set, nearer to Dh(P), we should define, given clause 4 above, two subsets for {Q} and {F}, $\underline{\{Q\}}$ and $\underline{\{F\}}$ respectively, such that:

$\underline{\{Q\}} == \{\langle t_1,t_2\rangle \mid \exists t_3: \langle t_1,t_2\rangle \in \{Q\}\ \underline{\text{and}}\ \langle t_2,t_3\rangle \in \{F\}\}$

$\underline{\{F\}} == \{\langle t_1,t_2\rangle \mid \exists t_3: \langle t_1,t_2\rangle \in \{F\}\ \underline{\text{and}}\ \langle t_3,t_1\rangle \in \{Q\}\}$

The previous sets can also be obtained as follows:

$\underline{\{Q\}} == \{Q\} \cap (\{Q\}\_1 \times (\{Q\}\_2 \cap \{F\}\_1))$

$\underline{\{F\}} == \{F\} \cap ((\{F\}\_1 \cap \{Q\}\_2) \times \{F\}\_2))$

Then, defining {P}_1 and {P}_2 we should consider $\underline{\{Q\}}$_1 and $\underline{\{F\}}$_2 instead of {Q} and {F}. Yet, although $\underline{\{Q\}}$ and $\underline{\{F\}}$ represent the set of tuples of Q and F, satisfying clause 4, because of the cartesian product X, {P} would still be greater than Dh(P); i.e.: denoting by $\underline{\{P\}}$ the set obtained by considering $\underline{\{Q\}}$ and $\underline{\{F\}}$ instead of {Q} and {F}, it is:

$Dh(P) \subseteq \underline{\{P\}} \subseteq \{P\}$

Thus $\underline{\{P\}}$ would still be an approximation of Dh(P).
Everything previously said about partial consistency, still holds. Yet $\underline{\{P\}}$ is less approximate than {P}.

The set expression $\underline{\{P\}}$, represents the least approximation we can get for the set of solutions of P, by the static analysis of its clausal definiton, following the approach of tracing variables. This happens just because the clausal definition of P was such that only two procedure calls shared a variable. In general if n procedure calls are such that each one shares one (or more), variable with others, then the least approximate solution need to be found by an iterative process:

1) First define each {Qi} for each procedure call; this would give a restriction of {Qi} in terms of other procedures with which Qi shares its arguments. Since the same is done for all procedure calls, there may be tuples of other procedures, satisfying the sharing conditions with Qi, which do not satisfy other sharing conditions in the same clause. This, in general, means that:
2) We need to define a {Qi}' identical to {Qi} but where the sets involved are the restricted ones, {F}'s instead of {F}'s.
3) The process of refining {Qi} has to go on until we get two sets, say {Qi}' and {Qi}", such that, either {Qi}'={Qi}", or else {Qi}" is empty.

This, informally described, refining process, will terminate. Let's in fact remind that, for all predicates Q and F, with two arguments (the same holds for predicate with any number of arguments), it is:

$\{Q\} \subseteq \{Q\}\_1 \times \{Q\}\_2,$ and also:

$\{Q\} = \{Q\} \cap (\{Q\}\_1 \times \{Q\}\_2)$ and

$(\{Q\}\_1 \cap \{F\}\_1) \times (\{Q\}\_2 \cap \{F\}\_2) \subseteq \{Q\}\_1 \times \{Q\}\_2$ and

$(\{Q\}\_1 \cap \{F\}\_1) \times (\{Q\}\_2 \cap \{F\}\_2) \cap \{Q\} \subseteq (\{Q\}\_1 \times \{Q\}\_2) \cap \{Q\} \subseteq \{Q\}$

Thus, the refining function is monotone descendent and it stops, either producing an empty set, {}, or producing the same set. Moreover, when the process stops, all sets such as {Q} and {F}, represent, the exact set of tuples satisfying all conditions in the clause. Than the set of solutions for the procedure defined by the clause in object, should be build in terms of these last sets.

The above process can be defined, perhaps more clearly, if we consider all variables, either locals or not, involved in a clause. For each variable we define a set expression representing its possible values, deducing such set expression from the And/Or graph of the clause. Thus, given a clause such as:

P(X1, X2,...Xn)<- Q1(t11,..tm1),..., Qn(tn1,....tnk)

Let's denote by X the set X1,X2,..,Xn and by Y the set Y1,Y2,...Yw of local variables in the previuous clause. Then for some of the above tij it is either

tij $\subseteq X$, or Tij $\subseteq Y$.

Consider the And/Or graph, G, correspondent to the above clause, and trace all variables in it. Then for each variable Zi, with Zi $\in \mathcal{X}$, or Zi $\in \mathcal{Y}$, consider the subgraph GZi of G which correspond to the trace of Zi, and trasform it as follows:

-the root is labelled by $\{\sigma 1 Zi\}$

-all And nodes become nodes; all Or nodes become U nodes;

-all nodes Qj(tj1,....tjm), are replaced by an node with k arcs leading out, respectively, to a nodelabelled by {Cj}_k; for Qj(tj1,....tjm) such that there exists a k= s:r, with $1 \leq s \leq r$ and tjk=Zi. Appendix 2 gives an example.

We can deduce $\{\sigma 1 Qj\}$, in the same way we deduced {Qj}, by defining such $\{\sigma 1 Qj\}$ in terms of {Cj} and $\{\sigma 1 Zi\}$, depending on the variables of Qj in the clause. Making sure that by i we always get a different set identifier, we can get a set of equations which can be transformed into some solved form, by a transformation process analogous to the one presented in next section. The set of equations in solved form, thus obtained, can be transformed again by the above mentioned refining function.

The set of equations we get by this approach can be summarized as follows: Given a set of clauses A,
- let $\mathcal{P}$ be the set of procedure symbols, defined, or just used, in clausal definitions of A;
- let $\{P1,P2,....Pn\} \in \mathcal{P}$, be the set of procedure defined in A;
then, for all Pi an equation is built which looks as follows:

$$\{Pi\} == \bigcup_{j=1}^{u} \left(\prod_{k=1}^{m} \{\sigma j X_{jk}\} \right) \quad \text{with } X_{jk} \in \mathcal{X}_{ij} \text{ and } \mathcal{X}_{ij} \text{ the set}$$

of variables, terms of Pi in the j-th clause.

Let $\mathcal{Z}_{ij}$ be the set of all variable symbols, local or not, in the j-th clause defineg Pi, then: For each variable symbol $Y_{jk} \in \mathcal{Z}_{ij}$, we have a set of equations as follows:

$\{\sigma j Y_{jk}\} == T$ where T is a set expression containing symbolic sets such as $\{\sigma j Cw\}$.

For each procedure Qw, called in the j-th clause defining Pi, we have a set of equations such as:

$$\{\sigma j Cw\} == \{Qw\} \cap \left(\prod_{s=1}^{r} \{\sigma j Y_{js}\} \right)$$

if r-1 is the number of argument of Cw and all Y_{js} are the variables, terms of Qw, in the j-th clause defining Pi.

4. TRANSFORMATION OF EQUATIONS

We will only consider the approach seen in 3.1, since we believe that the transformations we are going to define, can be accordingly modified to be applied to equations as defined in 3.2 above.

4.1 The transformation process

Thus, from 3.1 above, we have that: given a set of clauses A, Deduce(A), produces a set of equations, as in the monadic case, with the further complications of projections. As to transform the set of equations define in 3.1 above, let us observe that the best result we would like to get, is a a set of equations, each one of which associates a ground set (a set of constant symbols), to a procedure. Since some procedures may be defined in terms of undefined procedures, and since we believe that this is a useful information to keep, we want final set expressions, associated to procedures to mantain such references. Thus:

We say that a set of equations is in '*solved form*' if and only if, each equation has the form:

{Pi}==T or {Pi}j_k== T

for all procedure symbols Pi, and all integer j and k, and all set expression T such that:

1) T is a ground set; else
2) T is the empty set {}; else
3) T is a set expression which contains symbolic sets {Qj} and such that {Qj} does not appear on the left-hand side of any other equation (Qj is an undefined procedure).

We define now the following transformation algorithm

Transf1:
1) apply rule RR1;
2) apply ER, until possible;
3) apply S axioms, until possible;

Replacement Rule 1 (RR1)

For all equations of the form: $\{Pi\} == \bigcup_{j=1}^{n} \prod_{k=1}^{m} \{Pi\}j\_k$

replaces each occurence of '{Pi}_k', in all set expressions of all other equations, by

$$\bigcup_{j=1}^{n} \{P\}j\_k$$

Since from Deduce we get equations such as, for example:

{P}== ({P}1_1X .. X{P}1_m) U ......U ({P}u_1X .. X{P}u_m)

{P}1_1== ({M}_1 ∩ {T}_2) X ({P}_1 ∩ {D}_3)
...
{P}1_m==...........
...
...
{P}u_1==....
...
{P}u_n==........

{M}== ({M}1_1 X.. X{M}1_mm) U ...U ({M}k-1 X.. X {M}k_mm)
...
Rule ER1 allows to eliminate all references of type '{P}_w' and replaces them by a more detailed set expression in terms of '{P}j_w''s. By ER1, the previous example would be transformed in:
{P}== ({P}1_1 X .. X{P}1_m) U ......U ({P}u_1X.. X{P}u_m)

{P}1_1== ({M}_1 ∩ {T}_2) X (({P}1_1 U..U {P}u_1) ∩ {D}_3)
....
with all other equations modified accordingly.

Elimination Rule (ER)

Given an equation such as: {Pi}j_k== T
such that T contains ' {Pi}j_k', replaces '{Pi}j_k' in T, by the empty set {}.

Synthesis axioms (S)

-For all set expressions T:
{} U T == T
{} ∩ T == {}
-For all ground sets D_i, i=1,n : $\prod_{i=1}^{n} D_i$ is defined as usual;
Operations of U and ∩ are defined for ground sets as usual; moreover S will contain axioms for associativity and distributvity of U and ∩ .

Rule ER, defined for the monadic case, as been modified into rule ER above, to take into account projections of tuples. For the monadic case rule ER was an obvious consequence of the observation that, given a recursive clause such as:

P(X) <- Q(X),R(X),P(X)

according to its standard semantics, no denotations, different from those generated by all other clauses defining P, will be generated by that clause. In the n-adic case, the analogous happens for recursion over the same argument of a procedure. That is, consider the following clause:

P(X,Y)<-P(X,Z),R(Z,Y)

From the standard semantics we have that the previous clause adds tuples to the denotation of P, defined by the other clauses defining P, in the sense that it adds tuples where only the second elements may be new. Said it another way, considering the greatest set of solutions for P, denoted by: Dh(P)-1 X Dh(P)-2, the above clause may add elements to Dh(P)-2, not to Dh(P)-1. Thus, since we are now looking for the greatest approximation of Dh(P), we can consider the above clause having the same greatest set of solutions of P defined as:

P(X,Y)<-P'(X,N),R(Z,Y)

P(X,Y)<-P'(X,Y)

and where P' is defined as the rest of P. That is, if P was defined by u clauses and the one considered previously was the k-th, then P' is defined so that:

$$Dh(P') == \bigcup_{i=1}^{k-1} Dh(P)i \cup (\bigcup_{j=k+1}^{u} Dh(P)j$$

Thus we define RR so that each time we have equations such as:

{P}i_j= ({P}i_j U T1) X (...

we replaces all occurences of '{P}i_j', on the left hand side of the previous equation, by the empty set '{}'.

<u>Transf1</u> as defined above will certainly stops, since the number of substitutions RR1 has to do is finite; moreover, each substitution produces a set of equations such that the same type of sets {P}_k, for all predicates, will not appear anymore in anyone clause, unless P is undefined (thus no equation exists for {P}); RR1 needs to be done only once.
The same, of course, holds for RR;
S axioms are obviously convergent, and a point will be reached so that no one of them can be applied anymore.

Further transfromations are defined by the following algorithm:

<u>Transf2</u>

1) apply RR2;
2) apply RR;
3) apply S axioms;
4) repeat from 1) to 3) until all of them cannot be applied anymore.

<u>Replacement</u> <u>Rule</u> 2 (RR2)

Given an equations of type: {P}j_k= T
where T is a set expression; replaces each occurence of '{P}j_k' by T in all set expressions of all other equations.

RR and S axioms are define as in <u>Transf1</u>.

Rule RR2 is a transformation analogous to RR1.

<u>Transf2</u> stops, as well as <u>Transf1</u> does. Let's in fact observe that:
RR2 is applied after <u>transf1</u> is completed; no equation, such as {P}j_K=T, will be such that T contains '{P}j_k' itself (because of RR in <u>Transf</u>1);
At each step, RR2 eliminates all references to sets such as '{P}j_k'; thus, the next time RR2 won't be applied to the same equation; since this applies to all equations and since there are a finite number of equations, after a while, RR2 will not be applicable anymore.
Because of the same sort of considerations, about RR and S axioms, we can conclude that <u>Transf2</u> terminates, producing a set of equations each one of which has associated: either a ground set, or {} or else a set expression containing references to undefined procedures, i.e. it produces a set of equations in <u>solved</u> <u>form</u>.

At the end, since references to {P}j_k do not appear in any set expression and since they where built just for the sake of transformations, all equations for such sets can be eliminated; all that will remain is a set of equations, for the procedures defined in the set of clauses given in the first place.

5. FEW REMARKS ON TRANSFORMATIONS

The transformation process, presented in Section 4, is such that the following diagram commutes:

$$\begin{tikzcd}
DH(A) \arrow[dd, "\subseteq"'] & & A \arrow[ll] \arrow[r, "Deduce"] \arrow[dl] \arrow[dd, "Transf1"'] & EA \arrow[dd, "Transf1 + Transf2"] \\
& DH & & \\
DH(A') & & A' \arrow[ll] \arrow[ul] \arrow[r, "Deduce"'] & EA'
\end{tikzcd}$$

fig. 3

In fact, *Transf*1 and *Transf*2 steps, producing a set of equations, EA', in solved form; thus, the algorithm given by *Transf*1 followed by *Transf*2 is *complete*. It ensures a solution to the set of equations given in the first place.

Furthermore, the set of equations EA', can be deduced from a set of clauses A', such that A and A' have the same greatest approximate set of solutions, *DH*, with: *DH*={*Dh*(P1),....*Dh*(Pn)}, where each *Dh*(Pi) is the greatest set of approximate solutions of Pi, for all procedures Pi defined in A.
In fact, BR1, BR2 and S axioms, do not alter the semantics of the set of equations they are applied to; thus, the set of clauses correspondent to the set of equations, before and after BR1, BR2 and S axioms, can be obtained by a similar transformation of clauses in A.

The transformation process given in Sec. 4, is equivalent to the non-deterministic algorithm, given in |11|, which transform a given set of equations into another one, in soved form, to find an efficient unification algorithm. BR1 and BR2 are analogous to '*Variable Elimination*', |11-12|, and to the *Unfolding* transfromation defined in Program Transformations, |13-14|. ER is analogous to the transformation which erases equations such as x=x, in |11|, and Compaction in |12|. Also ER is equivalent to represent by {} the failure of transformations, |11| for equations: x=t, where t contain x.

For all procedures Pi, defined in A and A', their denotations, as defined by the standard semantics is such that:

$Dh_A(Pi) \subseteq Dh_{A'}(Pi)$ thus $DH(A) \subseteq DH(A')$

with DH(A), DH(A'), $Dh_A(Pi)$ and $Dh_{A'}(Pi)$ defined as for fig. 1, 2.

The above results is due to the method of not considering local variables at all, as it has been shown in Section 3.1.

6. CLAUSAL DEFINITIONS WITH MONADIC FUNCTIONS

In Section 2 we introduced a notation for functions, used in |10|. Now we are going to see how to extend results of previous sections 3-4, for clauses with functions.

Functions can be, either in terms of procedure being defined by a clause or else, be terms of procedure calls.
Let's first observe that:

P(f(X)) <- Q1(X),....Qn(X) is equivalent to:

P(f(X)<-P'(X)
P'(X)<-Q1(X),....,Qn(X)

Therefore, we can consider {P'} to be equivalent to {P} where the clausal definition of P does not contain any function, on its definition part. I. e.

$$\{P\}==\{f.\{P'\}\} \quad \text{and} \quad \{P'\}== \bigcap_{i=1}^{n} \{Ci\}$$

Thus for the above definition of P, we can deduce:

$$\{P\}==\{f.(\bigcap_{i=1}^{n} \{Ci\})\}$$

Thus, when functions are terms of a procedure being defined by a clause, the result of the procedure, relatively to that particular argument, in that particular clause, is given by:

{P}j_k== {f.T}

where T is derived in the same way as in 3.1, as if the function f didn't appear at all.

On the other hand if a variable X, argument of a procedure definition, is also the argument of a function 'g', in a procedure call to Q, then, instead of considering {Q}_k, we will consider: <u>dom</u>(g,{Q}_k). Which is a consequence of the meaning of <u>dom</u> (Section 2), and the consideration that:

P(X)<- Q(f(X)

is such that the solutions for P, will be all those values 'v', such that: f(v) ∈ Dh(Q), i.e. a set D such that '{f.D}' ∈ {Q}. For example, from:

P(f(X),g(Y),m(n(Z)))<- Q(X,h(Y)), R(m(X),Z) we have :

{P}1_1== {f.({Q}_1 ∩ <u>dom</u>(m,{R}_1))}
{P}1_2=={g.<u>dom</u>(h,{Q}_2)}
{P}1_3=={m.{n.{R}_2}}

The second approach in 3.2, needs to be slightly modified according to the previous notations.

For what concern transformations, rule RR1 and RR2 need to be modified so that replacements are not applied to references in non-atomic sets (i.e. sets such as {f.T}) and in arguments of the <u>dom</u> function. The solved form of set expressions is thus, such that symbolic sets, defined by other equations may appear only in set expressions which are part of non-atomic sets, or argument of the function <u>dom</u>. Everything previously said about transformations in Section 4, still holds.

At this point a further transformation can be done to expand a bit more set expressions in non-atomic sets, and in arguments of dom, to get some more information about the results of procedures, avoiding non termination of transformations, because of recursive set expressions. Although it will not be dealt with in this paper, we believe a notation, for non-atomic sets, can be found, such that, by a similar process of transformations, the fixpoint of such set expressions can be derived. We will then be able to represent data, built by recursive applications of functions, by a self-contained, recursive symbolic expression. For the moment we propose the following further transformations, for the set of equations obtained by the modified algorithms Transf1 and Transf2.

Transf3:

- apply RR1 so that replacemts take places in non-atomic sets and in set expressions, argument of dom.
- given a set of n equations, choose one of the form: {P}j_k==T;
1) replace each occurense of the left hand side of the given equation, by its right hand side, in all set expressions of other equations.
2) apply SS axioms;
3) choose an equation of the form {P}j_k==T, which has not been choosen yet;
4) repeat 1-3, until all equations have been choosen once.

Axioms SS (old S axioms plus axioms for non-atomic sets and dom expressions) are listed in Appendix 3. They can be proved convergent and consistent with the meaning of non-atomic sets and the dom function. The set expressions still represent approximate solutions of procedures.

7. CONCLUSIONS

We have considered logic programs in the Horn clauses form of logic with monadic functions. We have then presented two approaches to derive, from a given set of clauses a set of equations. The set of equations obtained represent, for each procedure, its greatest, approximate, set of solutions; i.e. a set which contains the denotation defined by the standard semantics.

Equations are derived from clauses by a data flow analysis, for the variables involved in the clause, carried on over the correspondent And/Or graph.

Given a set of equations (derived as in the first one of the two approaches presented), we define a transformation algorithm which reduces equations to a solved form. The set of equations thus obtained is such that each equation expresses, by a set expression, the set of approximate solutions for a given procedure. By approximate set of solutions for a procedures, we mean a set of values (when possible) some of which are correct solutions for the given procedures, while some others are wrong solutions. In any case, no other values, outside the approximate set of solutions, can be correct.

The aim of the paper is not to find tranformations in order to obtain more efficient programs, as it is the case for Program Transformations and Synthesis |13-14-15|. Our aim instead, is to find some properties of a program, from the static analysis of its definition, in the framework of Abstract Interpretations of Program, |6|. It is because of this that, for example, we believe that refences to undefined procedures should be kept in set expressions, for they could be used as a tool for program verification, program construction and composition of programs which have been defined separately.

REFERENCES

|1| **Kowalski, R.A.** - Predicate Logic as Programming Language, Proc. Information Processing 74, North Holland Pub. Co., Amsterdam, pp.569-574, 1974.

|2| **Van Emden, M. H.** and **Kowalski R.A.**- The Semantics of Predicate Logic as a Programming Language, Journal ACM, Vol. 23, No. 4, pp. 733-742, Oct. 1976.

|3| **Colmerauer, A.** et al. - Un Systeme de Communication Homme-Machine en Francais - Rapport preliminaire, Groupe de Researche en Int. Art., Universite d'Aix - Marseille - Luminy, 1972.

|4| **Warren,D., Pereira, L., Pereira, F.** - PROLOG: the language and its implementation compared to LISP, Proc. Symp. on A.I. and Prog. Lang., SIGPLAN Notices, Vol. 12, No. 8, and SIGART News. No. 64, pp. 109-115, Aug. 1977.

|5| **Clark,K. L.** and **McCabe, F.** - IC-PROLOG reference Manual, CCD Research Report, Imperial College, London, 1979.

|6| **Cousot,P., Cousot, R.**- Abstract Interpretation : A Unified Lattice Analysis of Programs by Construction or Approximation of Fixpoints. In : SIGACT/SIGPLAN Conf. Rec. of the Fourth ACM Symp. on Principle of Programming Languages; Los Angeles, Cal., jan. 17-19, pp. 238-252, 1977

|7| **Kowalski, R.A.** - Logic for Problem Solving; Artificial Intelligence series, (Nilsson, N.L. ed.), North Holland, 1979.

|8| **Levi, G., Sirovich, F.**- Generalized And/Or Graphs. Artificial Intelligence, 7, pp. 243-259, 1976

|9| **Marque-Puchen, G.**- Equations booleennes generalisees et Semantique des programmes en logique du premier ordre monadiques. PhD Thesis, Ecole Normale Superieure, Paris.

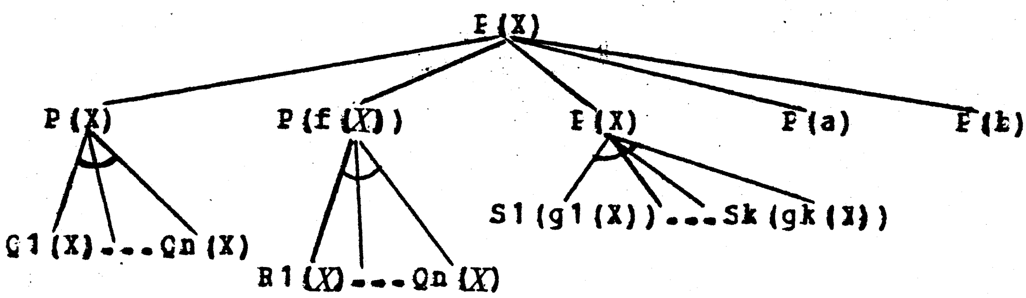
|10| **Asirelli, P.** - Horn Clauses Form of Logic: Algebraic Static Semantics of Programs. Int. Rep. I.E.I., B82-23, 1982.

|11| Martelli, A., Montanari, U.- An Efficient Unification Algorithm, ACM Trans. on Prog. Lang. and Systems, Vol. 4, n. 2, April 1982.
|12| Colmerauer, A.- PROLOG and Infinite Trees, in Logic Programming, K.L. Clark and S.-A. Tärnlund eds., Accademic Press, 1982.
|13| Burstall, R.M., Darlington, J.- A transformation system for developing recursive programs. Journal of the ACM 24, No. 1, 46-67, 1977.
|14| Clark, K.L., Darlington, J.- Algorithm classification through synthesis. The Computer Journal 23, No. 1, 1980.
|15| Burstall, R.M. : Recursive programs: Proof, transformation and synthesis. In "Rivista di Informatica" 7, pp. 25-42, 1976.

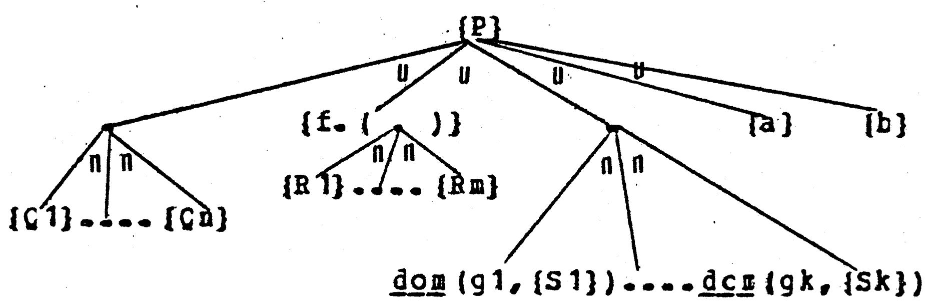
APPENDIX 1

P(X)<-Q1(X),....Qn(X)
P(f(X))<-R1(X),....Rm(X)
P(X)<-S1(g1(X)),....Sk(gk(X))
P(a)<- P(b)<-

Its correspondent And/Or graph can be drawn as follows:

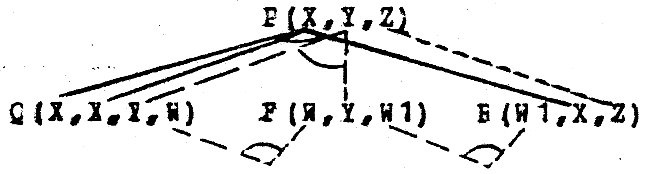


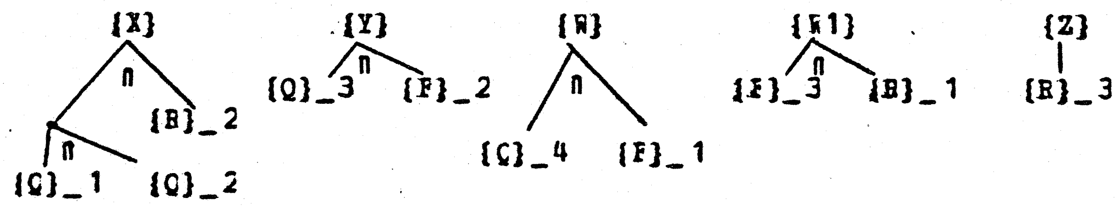
From the previous graph, we can deduce the following:



APPENDIX 2

P(X,Y,Z)<- Q(X,X,Y,W), P(W,Y,W1), R(W1,X,Z)





{P}== {X} X {Y} X {Z}

{X}== ({Q}_1 ∩ {Q}_2) ∩ {R}_2
{Y}== {Q}_3 ∩ {P}_2
{Z}== {R}_3
{W}== {Q}_4 ∩ {P}_1
{W1}== {P}_3 ∩ {R}_1

{Q}=={Q} ∩ ({X} X {X} X {Y} X {W})
{P}=={Q} ∩ ({W} X {Y} X {W1})
{R}=={R} ∩ ({W1} X {X} X {Z})

APPENDIX 3

1- For all set expressions A and B, such that A and B are ground sets:
- A U B == { x | x ∈ A or X ∈ B}
- A ∩ B== { X | x ∈ A and x ∈ B}

2- For all set expression D :

- D U {} == D
- D ∩ {} == {}
- D U {?} == {?}
- D ∩ {?} == D
- (D U B) ∩ D ≡ D ∩ (D U B) ≡D ∩ (B U D) == D
- (D ∩ B) U D ≡ D U (D ∩ B) ≡D U (B ∩ D) == D

3) For all atomic sets D, and all non-atomic sets H

- $D \cap H == \{\}$

4) For all non-atomic sets, and for all l,w and n:

- $\{f_1 . \{f_w \ldots \{f_n . \{\}\} \ldots\} == \{\}$

- $\{f_1 . D\} \cup \{f_1 . \{?\}\} == \{f_1 . \{?\}\}$

- $\{f_1 . D\} \cap \{f_1 . \{?\}\} == \{f_1 . D\}$

5) For all non-atomic sets and all set expressions H, D and all l, k, s:

- $\{f_l . H\} \cap \{f_k . D\} == \{\}$ iff $l \neq k$

- $\{f_l . \{f_k . H\}\} \cup \{f_l . \{f_s . D\}\} == \{f_l . (\{f_k . H\} \cup \{f_s . D\})\}$

- $\{f_l . \{f_k . H\}\} \cap \{f_l . \{f_s . D\}\} == \{f_l . (\{f_k . H\} \cap \{f_s . D\})\}$

-For all functions f:

$\underline{dom}(g, \{\}) == \{\}$

- and for all ground sets D:

$\underline{dom}(g, D) == \{\}$

-For all set expressions T:

$\underline{dom}(g, T) \cup \underline{dom}(g, T) == \underline{dom}(g, T)$
$\underline{dom}(g, T) \cap \underline{dom}(g, T) == \underline{dom}(g, T)$
$\underline{dom}(g, \{g.T\}) == T$

$\underline{dom}(g, \bigcup_{i=1}^{n} T_i) == \bigcup_{i=1}^{n} \underline{dom}(g, T_i)$

$\underline{dom}(g, \bigcap_{i=1}^{n} T_i) == \bigcap_{i=1}^{n} \underline{dom}(g, T_i)$