ON COMPILING PROLOG PROGRAMS ON DEMAND DRIVEN ARCHITECTURES.

Bellia M.(*), Levi G.(*), Martelli M.(+)

(*) Dipartimento di Informatica
University of Pisa (Italy)

(+) CNUCE Institute of CNR
Pisa (Italy)

ABSTRACT

A compiler is proposed that maps Prolog clauses into a language (LCA/1) with clauses annotated according to functional dependencies. LCA/1 has a demand driven computation rule and allows to cope with streams and lazy constructors.

The compilation eliminates the non-determinism related to the choice of the literal to compute and guarantees an efficient computation.

1. Introduction.

Non-determinism in Prolog comes in two flavours [1]. The first one is related to the full declarative programming style and comes from the absence of any ordering in the literals occurring both in the clause right-part and in the goal. The second one is related to the relational calculus and comes from the existence of superposable clauses (i.e. clauses whose left parts atomic formulas are unifiable).

Both of the above features contribute to making Prolog a milestone of the logic based programming languages and, at the same time, the basis for all the applications where calculus and reasoning merge: expert systems, relational knowledge base management, software systems specifications and various A.I. applications are only some of them [2,3,4].

Nevertheless, all these powerful Prolog aspects cause a high complexity in the Prolog run-time support because a non accurate choice of the literal to be computed can make highly non-deterministic even potentially deterministic computations. This is a direct consequence of the first type of non-determinism because Prolog programs do not explicitly state for each variable which literals "compute" the value and which literals use such a value.

Obviously, specific interpreters choose particular strategies such as the left to right evaluation of the literals, but this is a very strict choice and does not solve the problem. Incidentally, it is worth to note that this kind of complexity cannot be reduced by running programs on efficient and Prolog oriented machines.

In order to avoid the first type of non-determinism and to speed up the computation of those relations which are (multi-output) functions, many authors [5,6,7] have experimented control languages to attach algorithms to Prolog programs [8]. The authors have considered some logically based functional languages [9,10] and defined a functional logic language, LCA [11], which is a clause language with terms constrained to be either input or output terms. LCA could integrate Prolog, as an algorithmic component which allows to explicitly express programs involving functions and to compute them in a simple and efficient way.

Nevertheless, all the proposed solutions are partially inadequate. In fact all of them loose transparency with respect to declarativeness: i.e. the resulting programs contain procedural features.

Our aim is:

- to save the Prolog expressive power with its uniform view of relations and functions;
- to develop a technique for automatically eliminating the first type of non-determinism by attaching algorithms to clauses.
- to develop an efficient interpreter able to compute the intermediate form obtained with the above step.

The basic idea to achieve this goal is to define a language

(LCA/1, a generalization of LCA) whose programs are sets of "fully annotated" (Horn) clauses. Full annotation means that all the variables (not the terms) occurring in a clause (or goal) are annotated as INput or OUTput variables. Different occurrences of the same variable are possibly annotated in different ways. The "fully annotated" clauses must obey some syntactic constraints ensuring that each OUT variable can be computed in exactly one way.

The language interpreter has been defined along the lines of the interpreter already given for LCA. Its main features are:

- a demand driven computation rule;
- the ability to handle lazy data constructors;
- the ability to handle only the second (and really semantic) type of non-determinism.

The second step is to define a translator from Prolog programs into fully annotated programs. The translator associates to each clause of a Prolog program a set of fully annotated clauses. Each of them expresses both the specific state that the variables in a goal must satisfy in order to apply the clause (i.e. the variables which are already bound or not), and a specific functional dependency among the atomic formulas (i.e. which computes what). All the fully annotated clauses, associated to each clause, only depend upon the variables occurring in the clause and are not superposable.

The compilation of Prolog programs onto a demand driven machine seems a promising solution to save on one hand, all the features of Prolog programming and, on the other hand, to earn the efficiency of running programs on a demand driven architecture.

Section 2 will give a brief introduction to LCA/1. Sections 3 and 4 treat the translation in detail, while section 5 will describe the LCA/1 interpreter.

2. The LCA/1 language.

In this section we will not describe all the details of LCA/1, because it is quite similar to other proposals [11], but we will point out the main differences.

The first one is that in a term the occurrence of a variable symbol x is always annotated by IN or OUT. We call these terms fully annotated data terms and we refer to variables annotated by IN (OUT) as input (output) variables.

The atomic formula will contain only fully annotated data terms.

Let us introduce some definitions:

- constant term: a term without variables;
- input term: a term with input variables only;
- output term: a term with at least one output variable.

The following are examples of fully annotated clauses:

$$
\begin{array}{ll}
*(s(x_{IN}), y_{IN}, z_{OUT}) & \leftarrow\!\!- *(x_{IN}, y_{IN}, w_{OUT}), +(w_{IN}, y_{IN}, z_{OUT}) \\
*(s(x_{IN}), y_{IN}, z_{IN}) & \leftarrow\!\!- *(x_{IN}, y_{IN}, w_{OUT}), +(w_{IN}, y_{IN}, z_{IN}) \\
*(S(x_{IN}), y_{OUT}, z_{OUT}) & \leftarrow\!\!- *(x_{IN}, y_{OUT}, w_{OUT}), +(w_{IN}, y_{IN}, z_{OUT}) \\
REV(x_{IN} \cdot y_{IN}, w_{OUT}) & \leftarrow\!\!- REV(y_{IN}, z_{OUT}), APP(z_{IN}, x_{IN} \cdot nil, w_{OUT}) \\
REV(x_{IN} \cdot y_{OUT}, w_{IN}) & \leftarrow\!\!- REV(y_{OUT}, z_{IN}), APP(z_{OUT}, x_{IN} \cdot nil, w_{IN})
\end{array}
\qquad (*)
$$

where s and . are function symbols and *, +, REV and APP are predicate symbols.

The predicate * holds if the third argument is equal to the product of the first two arguments, and the predicate REV holds if the first argument is the reverse list of the second argument. Moreover, the intended meaning of the first clause of * is that, for any x and y, the result of the product of s(x) and y is the sum of y with the product of x and y, while the second clause of * means that for any triple of numbers x, y and z, z is the result of the product of s(x) and y if z is the sum of y with the product of x and y.

Examples of fully annotated goals are the following:

$\leftarrow\!\!-\ *(s(s(0)), S(0), x_{OUT})$

$\leftarrow\!\!-\ *(s(s(x_{IN})), s(y_{OUT}), z_{OUT}), +(s(s(0)), x_{OUT}, s(s(s(0))))$

$\leftarrow\!\!-\ REV(a.b.c.nil, z_{OUT})$

$\leftarrow\!\!-\ REV(a.x_{OUT}.c.nil, c.b.a.nil)$

The syntax of the language has to satisfy some constraints to have the desired properties. In the following, we assume familiarity with the terminology and the notation used in [1].

Let $M_{IN}(a)$ ($M_{OUT}(a)$) be the multiset of the input (output) variables of an atomic formula a.

Let H<-- a1,a2,...,an be a clause, where H is the conclusion atomic formula, the ai's are the atomic conditions, and all the atomic formulas are fully annotated (a1,a2,...,an can also indicate a goal).

Condition 1.

1.1) For each clause and for each ai, $M_{IN}(H) \cap M_{OUT}(ai) = \emptyset$.

1.2) For each clause or goal the multiset $\bigcup_{i \in [1,n]} M_{OUT}(ai)$ must be a set.

This condition ensures that every variable is computed in exactly one way by only one atomic formula.

Condition 2.

For each atomic formula ai in a clause or goal, each variable belonging to $M_{IN}(ai)$ must belong to $M_{OUT}(ak)$ (or to $M_{IN}(H)$ in the case of a clause), where ak is an atomic formula of the clause or goal such that $i \neq k$.

This condition forbids to have atomic formulas whose input variables do not occur as output variables of other atomic formulas.

Condition 3.

The multiset $M_{IN}(H)$ must be a set.

This condition is complementary to Condition 1 (about the uniqueness of the computations), and forbids to put conditions on the input variables of the conclusion atomic formula; i.e. the unification process does not need to control equality on the input variables. This allows to have a simple and (possibly) parallel unification algorithm.

Constraints on the values computed by different variables are allowed and efficiently handled by the primitive predicate EQp. The semantics of EQp corresponds to the Prolog assertion:

$$EQp(\underbrace{x,\ldots,x}_{p+1\text{-times}}) \leftarrow\!- \square \qquad (EQ1(x,x) \leftarrow\!- \square).$$

Note that because of Conditions 1,2 and 3 all variable symbols occurring in $M_{OUT}(H)$, must belong either to $M_{IN}(H)$ or to $M_{OUT}(ai)$, for some ai in the clause, or must not occur in the clause right part.

As a consequence, any output variable is either computed by one atomic formula only or must be considered bound to the set of all the terms of the Herbrand Universe.

LCA/1 is a generalization of LCA [11] mainly motivated by the compilation of Prolog clauses. Such a generalization is obtained by redefining the term structure and by relaxing some constraints of LCA. Nevertheless, the main properties of the LCA semantics are saved in the operational semantics of LCA/1. Thus, the definition of the LCA/1 interpreter is structurally similar to the one defined in [11]. Section 5 briefly analyses the external evaluation rule and the new formulation of the computation rule needed to handle full annotations.

3. The compiler.

The compiler from Prolog into LCA/1 is a mapping of clause structures of Prolog into LCA/1 ones.

This mapping is based on the concept of state of the computation, i.e. the state of the variables during the computation of the current goal: each variable can be already bound (totally or partially computed) or not. The variable can be considered, in the first case, as a possible input and, in the second case, as a possible output for an atomic formula.

A second aspect of the concept of state is related to the applicability of a clause. LCA/1 allows to explicitly define, for each conclusion atomic formula, which variables are assumed to be input (and thus must be bound to a value by the unification), and which variables are assumed to be output (and will have a value "computed" by the clause) at resolution time.

The first aspect of state is also present in Prolog (bound and unbound variables in the unification process).

The main idea of the transformation is that a Prolog atomic formula implicitly expresses a finite number of possible

different states (the second aspect) and this number combinatorially depends upon the number of variables occurring in the clause. A Prolog clause can then be mapped into a set of fully annotated clauses, each of them expressing a particular state.

As an example of the transformation, the three clauses in (*) are some of the eight fully annotated clauses defined by the following Prolog clause:

$*(s(x),y,z) \leftarrow\!- *(x,y,w),+(w,y,z)$.

Let us take the first clause of (*), i.e.:

$*(s(x_{IN}),y_{IN},z_{OUT}) \leftarrow\!- *(x_{IN},y_{IN},w_{OUT}),+(w_{IN},y_{IN},z_{OUT})$.

This clause explicitly defines a state of applicability, where the variables x and y must be bound and where the variable z is computed by the the clause itself.

4. The transformation.

In order to formally define the transformation from PROLOG programs into LCA/1 programs we will use the following simple structures.

DEFINITION 1 (variable sequence or sequence).
To each term t we can associate the variable sequence containing all the variable occurrences as found by a pre-order term traversing process.

As an example, $\langle x,y,x,z\rangle$ is the sequence associated to the term $f(x,g(y,x),z)$.
If t is a constant term, the sequence associated to t is the empty sequence. Let s be the sequence of length n associated to the term t, s[i] (or t[i]), for each $i \in [1,n]$, selects the i-th variable in s.
In the following, the concept of sequence will be generalized to atomic formulas by associating to each formula of the form P(t1,...,tk) the sequence obtained by concatenating the sequences s1,...,sk associated to the terms t1,...,tk respectively.

DEFINITION 2 (annotated sequence).
Let s be a sequence of length n, we define $\{IN,OUT\}^s$ as the set of all the annotated sequences generated by s.

The annotated sequence $v \in \{IN,OUT\}^s$ differs from s because, for each $i \in [1,n]$, v[i] is the variable s[i] annotated by IN or by OUT. We call v[i] an annotation for the variable s[i].
The set $\{IN,OUT\}^s$ contains exactly 2^n annotated sequences.

DEFINITION 3 (substitution sequence).
Let s be a sequence and v be an annotated sequence of the same length of s, we define a substitution as the pair (s,v).

DEFINITION 4 (substitution applicability).
Let t be a term and S be the substitution (r,v), we say that S is applicable to t iff r is equal to the sequence associated to t.
The application of S to the term t results in the term t' such that:

$$\forall i \in [1,n], \quad t'[i] = v[i],$$

if n is the length of s.

The transformation maps a clause c into a set U(c) of annotated clauses. It will be described in a two step process. First of all, given a clause c of the form H<--L, we compute the set U'(c) of partially annotated clauses. The clauses in U'(c) have all the variables occurring in H replaced by annotated variables. In the first step, the local variables of c (i.e. variables not occurring in the clause conclusion) are ignored.
The second step takes care of the local variables by providing the computation of a fully annotated clause for each clause in the set U'(c). In the same way, the second step is able to provide the transformation of a goal statement into the corresponding fully annotated goal.

4.1 The computation of U'(c).

Let c be the clause H<--l1,...,ln, the computation of U'(c) proceeds as follows:

1) Define $\{IN,OUT\}^s$, where s is the variable sequence associated to H.

2) Compute the subset $K \subseteq \{IN,OUT\}^s$ which contains all the annotated sequences having multiple occurrences of the same variable annotated by IN. Note that, the set K could be empty. The set is empty if and only if the sequence s does not contain multiple occurrences of the same variable.

3) $\forall r \in \{IN,OUT\}^s - K$, let (s,r) be a substitution. Compute H'<--L' U'(c) as follows:
   + H' is the atomic formula resulting from the application of (s,r) to H;
   + L' is the sequence l1',...,lm' such that:
     - n <= m, and
     - $\forall i \in [1,n]$, and for each variable x occurring both in li and in the sequence s, li' contains x annotated as follows:
       1) if x occurs in r annotated by IN, then each occurrence of x is replaced in li' by x_{IN}.
       2) if x occurs in r only annotated by OUT, then one of the following holds:
          a) $\exists j \in [1,n]$ such that $i \neq j$ and lj' already contains an occurrence of x_{OUT}. Then each occurrence of x is replaced in li' by x_{IN}.

b) $\forall j \in [1,n]$, such that $i \neq j$, lj' does not contain occurrences of x_{OUT}. Then
   1) if li contains exactly one occurrence of x, then the occurrence of x is replaced in li' by x_{OUT}.
   2) if li contains $p+1$ occurrences of x, then
      + the first occurrence of x is replaced in li' by x_{OUT} and all the other occurrences are replaced by different renamings of x annotated by OUT.
      + let $x1_{OUT},\ldots,xp_{OUT}$ be the above introduced renamings. Then
        $$EQp\ (x_{IN}, x1_{IN},\ldots,xp_{IN})$$
        is a <u>special</u> atomic formula lu' in L' for some $u \in [n+1,m]$.

4) $\forall r \in K$, we add to the set resulting from step 3) the clause

$$H'\text{<--}l1',\ldots,ln',\ldots,lh',\ldots,lm' \qquad (n <= h < m)$$

obtained as follows:
+ for each variable x occurring in r more than once, let $x1_{IN},\ldots,xp_{IN}$ be a renaming for each occurrence but the first. Then
  $$EQp\ (x_{IN}, x1_{IN},\ldots,xp_{IN})$$
  is an atomic formula lu' for some $u \in [h+1,m]$
+ let r' be the annotated sequence r whose variables are renamed according to the above step, then (s,r') is still a substitution and $H'\text{<--}l1',\ldots,ln',\ldots,lh'$ is the result of step 3) applied to (s,r').

4.2 Example.

Let us consider the clause c:

$$A(x,d(x)) \text{ <-- } B(x,y),E(x,x)$$

where d is a function symbol, A,B and E are predicate symbols and x, y are the variables occurring in the clause such that y only is local. Then, the computation of $U'(c)$ proceeds as follows:

1) $s=\langle x,x\rangle$
   $\{IN,OUT\}^s = \{\langle x_{IN},x_{IN}\rangle,\langle x_{IN},x_{OUT}\rangle,\langle x_{OUT},x_{IN}\rangle,\langle x_{OUT},x_{OUT}\rangle\}$

2) $K=\{\langle x_{IN},x_{IN}\rangle\}$

3) $\forall\ s \in \{\langle x_{IN},x_{OUT}\rangle,\langle x_{OUT},x_{IN}\rangle,\langle x_{OUT},x_{OUT}\rangle\}$
   + $s=\langle x_{IN},x_{OUT}\rangle$
     $H'=A(x_{IN},d(x_{OUT}))$
     $L'=B(x_{IN},y),E(x_{IN},x_{IN})$
   + $s=\langle x_{OUT},x_{IN}\rangle$
     $H'=A(x_{OUT},d(x_{IN}))$
     $L'=B(x_{IN},y),E(x_{IN},x_{IN})$
   + $s=\langle x_{OUT},x_{OUT}\rangle$

$H' = A(x_{OUT}, d(x_{OUT}))$
$L' = B(x_{OUT}, y), E(x_{IN}, x_{IN})$
or
$L' = B(x_{IN}, y), E(x_{OUT}, x1_{OUT}), EQ1(x_{IN}, x1_{IN})$

4) $\forall r \in \{<x_{IN}, x_{IN}>\}$
+ $EQ1(x_{IN}, x1_{IN})$
+ $r' = <x_{IN}, x1_{IN}>$
$H' = A(x_{IN}, d(x1_{IN}))$
$L' = B(x_{IN}, y), E(x_{IN}, x_{IN}), EQ1(x_{IN}, x1_{IN})$

The computation defines two U'(c), each one containing four fully annotated clauses, which differ in the right part of the clause obtained from the substitution $s = <x_{OUT}, x_{OUT}>$.

4.3 Remarks about U'(c)

Proposition 1
For each Prolog clause c, U'(c) contains at least one clause. Moreover, U'(c) contains exactly the clause c iff the conclusion atomic formula of c has no variables.

Proposition 2
U'(c) as computed by steps 1)-4) is not unique. In fact, step 3.2) could lead to more than one U'(c), if more than one atomic formula in the right part contains a variable which, in the sequence r, is only annotated by OUT.
Actually, we are not concerned with the choice of U'(c), although the problem of choosing the best atomic formula is the key issue for optimization.

Proposition 3
The following properties hold for the annotations of the clauses in U'(c):

Property 1 No conclusion atomic formula contains more than one occurrence of the same variable annotated by IN, as guaranteed by the subset K in steps 2) and 4).

Property 2 No clause right part contains more than one occurrence of the same variable annotated by OUT, as guaranteed by step 3).

Property 3 For each clause whose conclusion atomic formula contains a variable annotated by OUT, only one of the following cases holds:
1-the same variable annotated by IN occurs in the conclusion atomic formula also;
2-the same variable annotated by OUT occurs in exactly one atomic formula in the clause right part;
3-the same variable annotated by OUT occurs in the conclusion atomic formula only.
This property is guaranteed by the variable renamings

introduced in point 2.b.2 of step 3).

Property 4 For each variable annotated by IN in a clause atomic formula, only one of the following cases holds:
- the same variable annotated by IN occurs in the clause conclusion;
- the same variable annotated by OUT occurs in exactly another atomic formula of the clause right part.

This property is guaranteed by point 1) and 2.a) of step 3).

4.4. The computation of U(c)

The computation of U(c) provides the annotation of the local variables occurring in c. Local variables are variables which occur only in the right part of the clause. Such variables are left unchanged by the computation of U'(c). Thus the following property holds:

Proposition 4
$\forall \bar{c} = H \leftarrow\!- L$, $\bar{c} \in U'(c)$ iff there exists L' such that:
$H \leftarrow\!- L' \in U(c)$.

Thus, in order to obtain U(c), for each clause $\bar{c}$ of U'(c), only L' has to be computed.

Let $H \leftarrow\!- l1,\ldots,ln$ be a clause in U'(c), then U(c) contains $H \leftarrow\!- l1',\ldots,lm'$ $(n<=m)$ such that:

5) $\forall i \in [1,n]$ such that li is already a fully annotated atomic formula (i.e., li does not contain local variables) then $li'=li$;

6) Let $i \in [1,n]$ be such that li contains at least a local variable x, then one of the following cases holds:
   1) $\exists j \in [1,n]$, such that $i \neq j$ and lj' contains an occurrence of x_{OUT}. Then each occurrence of x is replaced in li' by x_{IN}.
   2) $\forall j \in [1,n]$, such that $i \neq j$, lj' does not contain occurrences of x_{OUT}, then:
      a) if li contains exactly one occurrence of x, then the occurrence of x is replaced in li' by x_{OUT}.
      b) if li contains p+1 occurrences of x, then the following steps are performed:
         - the first occurrence of x is replaced in li' by x_{OUT} and all the other occurrences by renamings of x annotated by OUT.
         - let $x1_{OUT},\ldots,xp_{OUT}$ be the above introduced annotated renamings for x. Then
           $EQp(x_{IN}, x1_{IN},\ldots,xp_{IN})$
           is the atomic formula lu', for some $u \in [n+1,m]$, added to the right part of the transformed clause.

4.5 Example

As an example of computation, let us consider the computation of U(c1) and U(c2) in the case of the following predicates for the addition:

c1: $+(0,y,y)$ <--
c2: $+(s(x),y,s(z))$ <-- $+(x,y,z)$

where s is the successor function.

$$U(c1)=\{+(0,y_{IN},y1_{IN}) \leftarrow EQ1(y_{IN},y1_{IN}) \quad (1)$$
$$+(0,y_{IN},y_{OUT}) \leftarrow \quad (2)$$
$$+(0,y_{OUT},y_{IN}) \leftarrow \quad (3)$$
$$+(0,y_{OUT},y_{OUT}) \leftarrow \} \quad (4)$$

$$U(c2)=\{+(s(x_{IN}),y_{IN},s(z_{IN})) \leftarrow +(x_{IN},y_{IN},z_{IN}) \quad (5)$$
$$+(s(x_{IN}),y_{IN},s(z_{OUT})) \leftarrow +(x_{IN},y_{IN},z_{OUT}) \quad (6)$$
$$+(s(x_{IN}),y_{OUT},s(z_{IN})) \leftarrow +(x_{IN},y_{OUT},z_{IN}) \quad (7)$$
$$+(s(x_{IN}),y_{OUT},s(z_{OUT})) \leftarrow +(x_{IN},y_{OUT},z_{OUT}) \quad (8)$$
$$+(s(x_{OUT}),y_{IN},s(z_{IN})) \leftarrow +(x_{OUT},y_{IN},z_{IN}) \quad (9)$$
$$+(s(x_{OUT}),y_{IN},s(z_{OUT})) \leftarrow +(x_{OUT},y_{IN},z_{OUT}) \quad (10)$$
$$+(s(x_{OUT}),y_{OUT},s(z_{IN})) \leftarrow +(x_{OUT},y_{OUT},z_{IN}) \quad (11)$$
$$+(s(x_{OUT}),y_{OUT},s(z_{OUT})) \leftarrow +(x_{OUT},y_{OUT},z_{OUT})\} \quad (12)$$

Note that U(c1) and U(c2) are unique.

4.6. Remarks about U(c).

Because of Proposition 4, some properties, already given for the set U'(c), hold for the set U(c) as well. In the following, we state the properties which hold for the set U(c) and we show how the clauses in U(c) satisfy the conditions given for the annotated clauses of LCA/1.

Proposition 1'
Proposition 1 holds in the case of U(c) also. However, U(c) could contain exactly one clause c', such that $c \neq c'$, depending on the occurrence of local variables in c.

Proposition 2'
Proposition 2 holds in the case of U(c) also. In fact, in addition to the non-uniqueness of U'(c) (caused by step 3.2), step 6) could hold for more than one U(c) for similar motivations. The remarks given about U'(c), concerned with the choice of the best atomic formula, apply to U(c) as well.

Proposition 3'
Properties of U'(c), involving only the clause conclusion, obviously hold even for U(c), namely properties 1 and 3 of Proposition 3. In addition, clauses in U(c) satisfy Properties 2 and 4 because of steps 5) and 6).

We will now show that, if Proposition 3' holds, the conditions given for the clauses of the language introduced in Section 2 are satisfied.

- Condition 1 Point 1 is achieved by property 4' of Proposition 3', because, when it is applied to the conclusion atomic formula, the first case holds. Point 2 is guaranteed by property 2' of Proposition 3'.

- Condition 2 The condition immediately follows from property 4' of Proposition 3'.

- Condition 3 The condition is guaranteed by property 1' of Proposition 3'.

As a final remark let us note that property 3 of Proposition 3 is not essential and follows directly from the other properties in the Proposition.

Finally, a few words about the goal. A goal is a special clause structure whose left part is "empty", and, thus, it only has local variables. The computation of U'(c), in the case of a goal c, is the set {c}. Given U'(c)={c}, the computation of U(c) proceeds as in the case of any other clause structure. It results in the set {c'} whose unique clause is a fully annotated goal and satisfies all the above propositions.

4.7 Example

As an example of a goal computation let us consider the following clause c:

<-- +(3,u,v)

The computation of c is:

$$U(c) = \{\text{<-- } +(3, u_{OUT}, v_{OUT})\}$$

Note that the solution is unique.

5. The language interpreter.

While mentioning the language features, we pointed out in Section 2 how LCA/1 is a generalization of LCA, proposed for functional (even if non-deterministic) computations in Prolog-like programming enviroments. Thus the language interpreter we propose is defined along the same lines of the LCA interpreter given in [11]. It has a similar algebraic definition and it handles some features, like lazy constuctors and streams, in exactly the same way.

Nevertheless, some relevant differences must be considered, mainly with respect to:

- the evaluation order of the goal atomic formulas;
- the clause unification mechanism.

5.1 The evaluation order and the demand driven rule.

The evaluation order of atomic formulas in a fully annotated goal is established on the basis of a demand driven rule.

Each fully annotated goal contains some of the following three types of atomic formulas:

1) constant formulas: atomic formulas whose terms are only constant terms;

2) input formulas: atomic formulas whose terms are either constant or input terms and contain at least one input term;

3) output formulas: atomic formulas containing at least one output term.

The first two types of formulas correspond to formulas which only put constraints on the goal or on the values of the variables occurring in the goal. As a matter of fact, atomic formulas, whose predicate symbol is EQp, are of the second type and their evaluation constraints the evaluation of the formulas which use the same variables. Annotations allow us to define global each variable annotated by OUT which occurs in a goal and such that:

+ the variable does not occur annotated by IN in the goal
  or
+ the variable occurs annotated by IN in input formulas only.

Thus a goal could be partitioned into two parts. One part consists of the set of all the atomic formulas which contain at least one occurrence of a global. This part provides the computations of the "results" of the goal evaluation.

The atomic formulas of the second part do not contain globals and only provide the computations of intermediate (and, possibly unessential) values.

The evaluation of a goal proceeds as follows: the constant formulas are evaluated first, then the input formulas containing globals are considered. Finally, when the goal does not contain any constant nor input formulas with globals, the output formulas which contain at least one global are evaluated.

The evaluation of an atomic formula of the second or third type could require the evaluation of output formulas including atomic formulas not containing globals. In the case of the evaluation of formulas not containing globals, input formulas are evaluated first.

Note that the order is statically defined by the input-output relation among atomic formulas. The relation is induced by the occurrence of the same variable annotated by IN and OUT respectively in different atomic formulas. The relation we have defined is a partial order. Hence the choice of the formula, where more than one choice is possible, is unessential to a right sequentialization of the computation.

5.2 The clause application mechanism

The clause application mechanism allows to apply a clause to an atomic formula in the goal, and results in the evaluation of goal atomic formulas. Whenever the value of a variable is needed to apply a clause, the atomic formula computing that variable is selected (by the Demand Driven Rule) for the evaluation.

The mechanism is mainly based on a term unification mechanism which provides:

- the binding of the input variables which occur in the conclusion atomic formula of the clause with the corresponding input or constant terms of the goal atomic formula;

- the binding of the output variables which occur in the goal atomic formula with the corresponding output or constant terms of the conclusion atomic formula of the clause.

Thus, the application of the unification to terms is not symmetric. In fact, unification behaves, on one hand, like a match of input terms in the goal atomic formula to input terms in the clause conclusion, and, on the other hand, like a match of output terms in the clause conclusion to the output terms in the goal atomic formula.

The unification of a term in the goal atomic formula, tg, with the corresponding term in the clause conclusion, tc, has the following properties.

Proposition 5

The term tg is unifiable with tc if one of the following cases holds:

1) tg is a constant term;
2) tg is an input term and tc is either an input or a constant term;
3) tg is an output term and tc is either an output or a constant term.

Proposition 6

The unification of tg and tc results in the pair of unifiers $(\lambda_{IN}, \lambda_{OUT})$ respectively for input and output variables, if and only if:

1) tg is a constant term and λ_{IN} is such that:

$$tg = [tc]_{\lambda_{IN}}$$

Note that, if tc is an output term, the unification requires the evaluation of the right part of the clause in order to compute the output variables occurring in tc.

2) tg is an input term and λ_{IN} is such that:

$$tg = [tc]_{\lambda_{IN}}$$

In this case, the unification could require the evaluation of the goal in order to compute the variables occurring as inputs in tg and corresponding to terms (different from variables) in tc.

3) tg is an output term such that:

3.1) tg is an output variable. Then

$$[tg]_{\lambda_{OUT}} = tc$$

that is, λ_{OUT} contains a binding of the variable tg to the term tc. Moreover, tc must be a constant term or an output term containing only output variables.

3.2) tg is a term of the form f(tg1, ..., tgk) (where f is a data constructor and at least one of the tgi's is an output term), then:

$$[tg]_{\lambda_{OUT}} = [tc]_{\lambda_{IN}}$$

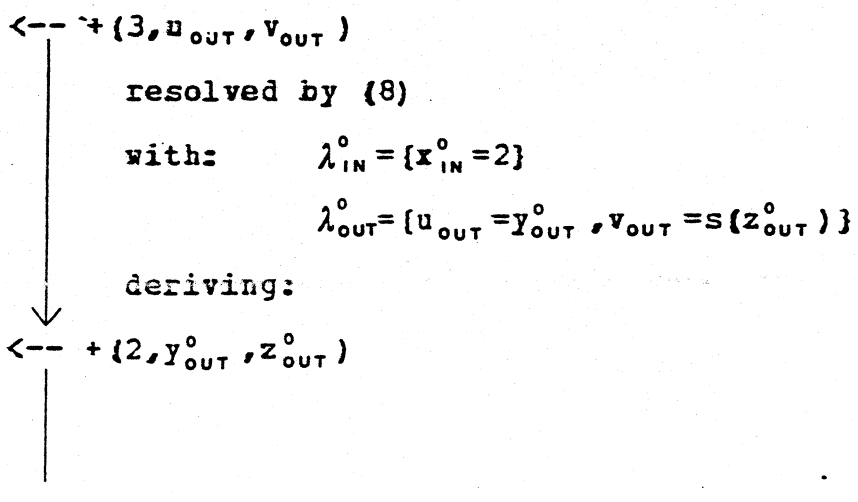
If this is the case and if tc is an output variable, the unification needs the evaluation of the right part of the clause to obtain for tc the term f(tc1, ..., tck). Then, the unification proceeds through the unification of tgi, tci for each i from 1 to k.

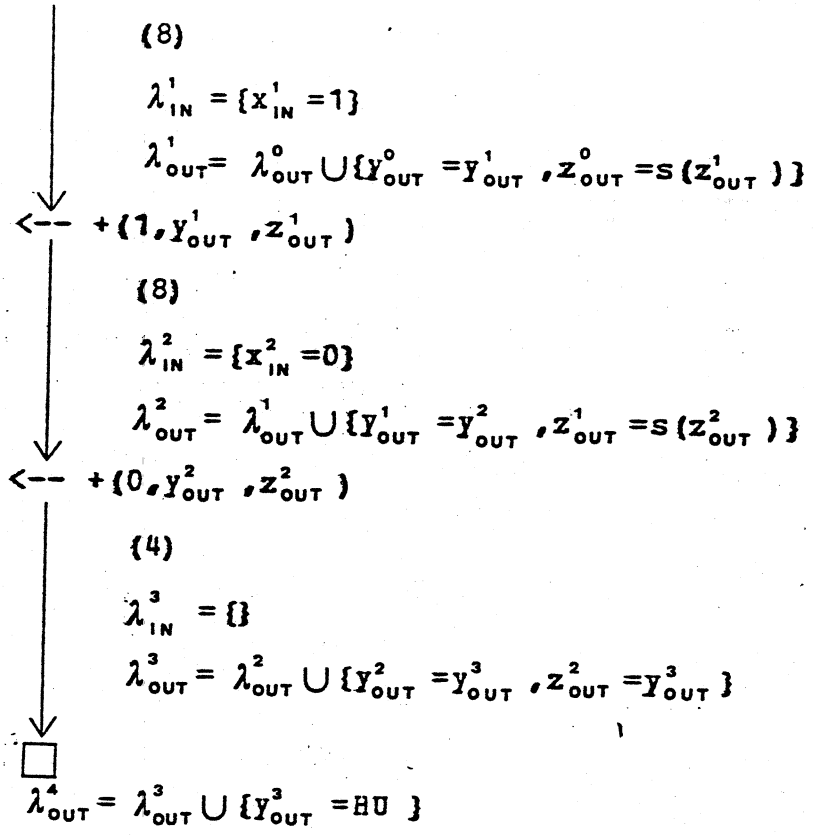
Note that, because of 3.1, if tc is an output variable, its value f(tc1, ..., tck) contains output variables only. Thus, to obtain an unification, tg must also be a term containing output variables only.

A special case arises when tc is an output variable which does not occur in the right part of the clause, i.e. there are no atomic formulas in the goal which can compute values for the variable. In this case the variable is considered bound to all terms of the Herbrand Universe, and the value of the variable is denoted by HU. The match of such a variable to an output term tg must bind the variables in tg to HU also.

5.3 Example

As an example of a computation of a fully annotated program, let us consider the evaluation of the goal in the example in 4.7 with the clauses U(c1) and U(c2) in 4.5.





6. Conclusion

The design of new machines for logic based languages, including the functional ones, requires the project of unconventional architectures oriented to efficiently handle the language computation rules.

Thus it is important to define a (small) nucleus of primitive rules which, on one hand, guarantees to express each language computation step and, on the other hand, becomes a model to tailor the language architecture.

In this trend, we have considered the selection of atomic formulas in the goals of a Prolog computation. As a matter of fact, the selection has a remarkable relevance in the Prolog implementation because:

- the selection affects the efficiency of the computations: i.e. it can cause too long computations;
- the selection requires a specific mechanism which can even affect the efficiency of the mechanism to handle the non-determinism.

Actually, the selection is handled in two different ways. The first, common to all the Prolog implementations, makes a static selection. This is achieved either by ordering the atomic formulas from left to right [12], or by using annotations [5]. The former does not cope with efficiency, while the latter looses the declarative transparency and does

not guarantees efficiency.

The dynamic handling of the selection is the second approach [13]. It allows efficient computations but requires mechanisms which are complex and hard to build.

A promising solution to this problem seems to be a compilation of the Prolog clauses into fully annotated clauses.

An annotation assigns a role to atomic formulas by distinguishing between the one which, for a given variable, must compute a value and the ones which will use that value. In this way, a functional dependency is statically imposed on the atomic formulas. Then the selection is handled by means of a demand-driven mechanism.

In addition to it, the proposed compilation allows us to reduce both the overhead of the unification mechanism (which becomes a matching mechanism) and of the computation environment (only the output terms unifiers, λ_{OUT}, must be kept).

However, some open questions can be considered.

The first is the choice of the object program when more than one is possible. The choice is semantically unessential (as we will point out in the following) and does not affect the design or the efficiency of the demand-driven mechanism. However, it is essential in order to shorten the computations.

Given a set S of Horn clauses, the choice solutions are strictly related to a selection function which guarantees, for each goal for S, a derivation (if any) with the smallest number of input clauses [14].

The use of partially annotated clauses (clauses like those occurring in U'(C)) together with the results concerning the superposition [15] seems a promising approach towards the definition of such a function.

For what the semantics is concerned, it is simple to prove that any object obtained by the compilation is semantically equivalent to the original Prolog set of clauses (source program).

The proof could ignore the problems arising from superposable clauses and show that the derivation of the fully annotated clauses is a special case of the LUSH resolution applied to Horn clauses.

Finally, the programming environment, the proposal allows to define, deserves some remarks.

Programming applications often need to integrate declarative programming with procedural one. Such an integration will allow to easily combine declarative and procedural knowledge (i.e. algorithms) and is currently been pursued by several projects, notably Robinson's LOGLISP [16].

To obtain it, attention has to be put on the integration level which must allow, on one hand, to easily merge declarative with procedural computations, and on the other hand, to maintain, as small as possible, the nucleus for the different types of computation.

LCA/1 seems a good candidate for the integration level, in particular it allows the same nucleus to compute both declarative and procedural programs. Moreover, the proposed compiler could be lightly modified in order to be applied to programs of partially annotated clauses, thus including pure Prolog programs, LCA programs and programs whose clauses contain both Prolog and LCA atomic formulas.

REFERENCES

[1] Kowalski, R.A. Predicate Logic as a Programming Language. Information Processing 74, North Holland, 1974, pp. 556-574.

[2] Kowalski, R.A. Logic for Problem Solving. Artificial Intelligence Series, N.J. Nilsson Ed., North Holland, 1979.

[3] Logic Programming. Clark K.L. and S.A. Tarnlund Eds., Academic Press, 1982.

[4] Proceedings of the 1st Int'l Logic Programming Conference. Marseille, 1982.

[5] Clark, K., McCabe, F. and Gregory, S. IC-PROLOG Language Features. In [3], pp. 253-266.

[6] Gallaire, H. and Lasserre C. Metalevel Control for Logic Programs. In [3], pp. 173-185.

[7] Pereira, L.M. and Porto, A. Intelligent Backtracking and Sidetracking in Horn Clause programs - The Theory. Departamento de Informatica, Universitade Nova de Lisboa, Rep. 2/79 CIUNL, October 1979.

[8] Kowalski, R.A. Algorithm=Logic+Control. Comm. of A.C.M., 22, 1979, pp. 424-431.

[9] Bellia M., Degano P. and Levi G. The call by name semantics of a clause language with functions. In [3], pp. 281-298.

[10] Bellia, M., Dameri, E., Degano, P., Levi, G. and Martelli, M. Applicative Communicating Processes in First Order Logic. Lecture Notes in Computer Science, 137, Springer Verlag, 1982, pp. 1-14.

[11] Bellia, M., Dameri, E., Degano, P. Levi, G. and Martelli, M. A Formal Model for Demand-driven Implementations of Rewriting Systems and its Application to Prolog Processes. I.E.I. Internal Report, IEI-B81-3, 1981.

[12] Moss, C. The comparison of several Prolog systems. Proc. of First Logic Programming Workshop, Debrecen, 1980, pp. 198-200.

[13] Pereira, L.M. and Porto, A. Selective Backtracking. In [3], pp. 107-116.

[14] Hill, R. LUSH-Resolution and its compliteness. DCL Memo No. 78, Univ. of Edimburgh, 1974.

[15] Sato, T. and Tamaky, H. Enumeration of success patterns in Logic Programs. To be presented at 10th ICALP.

[16] Robinson, J.A. and Sibert, E.E. LOGLISP: an alternative to PROLOG. Machine Intelligence No. 10, 1982, pp. 399-420.