Checking and Debugging of Two-level Grammars

Sadegh SAIDI and Jean-Francois BOULICAUT

Ecole Centrale de Lyon
Département Mathématiques-Informatique-Systèmes
BP 163
F-69131 Ecully cedex
e-mail : saidi@cc.ec-lyon.fr

Abstract. An extension of PROLOG which supports software specification by means of a class of two-level grammars is presented. AFFLOG logic programs are typed and modes can be specified if desired. By examining their underlying grammatical properties, a static analysis is performed. Our purpose is to support translator writing starting from a grammatical model that has been checked and debugged.

1 The context

Data processing can be viewed fundamentally as the transformation of data delivered as character strings into results which are character strings too. Therefore, programs should be designed as translators of the language defining the input data into the language defining the output data. Processing steps are considered as semantic computations on some intermediate languages.

Typical approaches of the so-called *grammatical programming framework* have been described e.g. in [Hehner-83,Torii-84]. However, these works are limited from the following points of view :

- the grammatical formalisms which are used. In general, only the context-free approximation of the input language is formalized.
- the static checks which are performed on the definitions. Few works emphasize the need to check and debug the specifications.

Our goal is to propose a powerful grammatical formalism (Chomsky's class 0) that helps to write readable specifications which can still be machine-oriented and automatically evaluated (e.g. to produce parsers or translators). We investigate the use of two-level grammars and in particular the Extended Affix Grammar formalism (EAG) [Watt-74].

In [Boulicaut-92], a Wide Spectrum Grammatical Programming Framework is proposed. Starting from formal specification by means of an EAG, a program (i.e. a translator) is considered as a device that computes tuples from a characteristic relation of this EAG. In this framework, the refinement of the specification, from prototyping through efficient implementation relies on affix grammars. A compiler compiler like STARLET [Beney-90] is used when we have to compute deterministic translations (a realistic need).

Besides, there is a close relationship between context-sensitive grammar processing and logic programming [Kowalski-79,Colmerauer-75,Deransart-88]. Chomsky's class 0 languages can be defined by logic grammars such as Metamorphosis Grammars [Colmerauer-75] or Definite Clause Grammars (DCG) [Pereira-80], which are easily compiled into PROLOG programs.

However, experiences show that a PROLOG translator writer needs to define "metastructures" to improve the readability of written translators and to check well-formedness conditions. Even during the prototyping phase, efficiency has to be improved by a backtracking control and an informed use of unification (taking concatenation associativity into account...). The production of more reliable PROLOG programs is made easier by using systems which have explicit type definitions. Two-level grammars like EAG provide a mean of expression for these metastructures and their relationships with PROLOG are well-known [Maluszynski-82a].

The main goal of our current research is to support the specification process by means of EAG. J. Maluszynski studied the question of programming with transparent W-grammars [Maluszynski-82ab,84] and the first experimental implementation of a logic programming language that should support these ideas [Maluszynski-82c,Näslund-87]. Then, our purpose is to explore the methodological impact of multiple transfers between grammatical and logic programming using EAG instead of transparent W-grammars.

From a methodological point of view (see Fig. 1), our approach aims to take the most of grammatical modelling and logic programming. There are three important relations during the life-cycle of a program : the *intended relation*, (the intended declarative semantics), the *specified relation* (what is stated by the specification) and the *computed relation* (what the programming system computes). A program design method must supply systematic design elements for these three relations, in particular help to debug computed and specified relations.

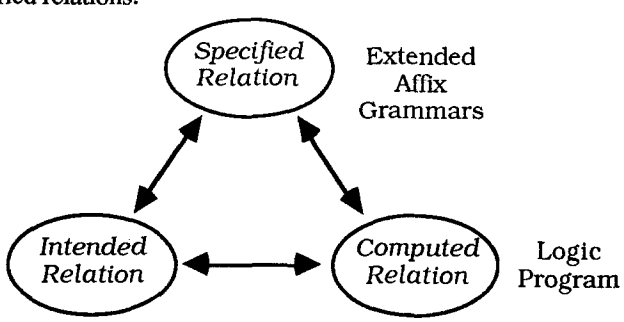

Fig 1 : A methodological point of view

Since programs are considered as translators, the specification method relies on techniques borrowed from compiler design (the specified relation is described by an EAG) while the relation is computed through logic programming. We also write logic programs which do not generate a language (the specified relation is described by Horn clauses and the language generated by the underlying grammar is reduced to the empty string). Informations can be extracted from these logic programs provided one consider them as two-level grammars.

As a first step towards computer-assisted grammatical modelling, we propose the experimental tool AFFLOG.

AFFLOG can be considered to be :

- A two-level grammatical formalism. A text is a two-level grammar which borrows concepts from extended affix grammars and logic grammars. Thus, such a text will generate a language. The AFFLOG formalism aims to bridge the gap between the informal specification of a problem and its formal one by enhancing readability (the use of strings instead of compound terms).
- An extension of PROLOG. A PROLOG program is a special case of an AFFLOG program where a (meta) grammar specifies the structure of the terms and where the terminal vocabulary is empty (i.e. its language is reduced to the empty string).

2 The AFFLOG Programming Tool

2.1 An overview

An AFFLOG program mainly consists of a metagrammar (CF-rules describing intermediate languages i.e. the syntax of the terms which will be used) and a hyper-grammar which is the definition of a context-sensitive grammar.

Figure 2 illustrates the processing done by the AFFLOG logic programming system.

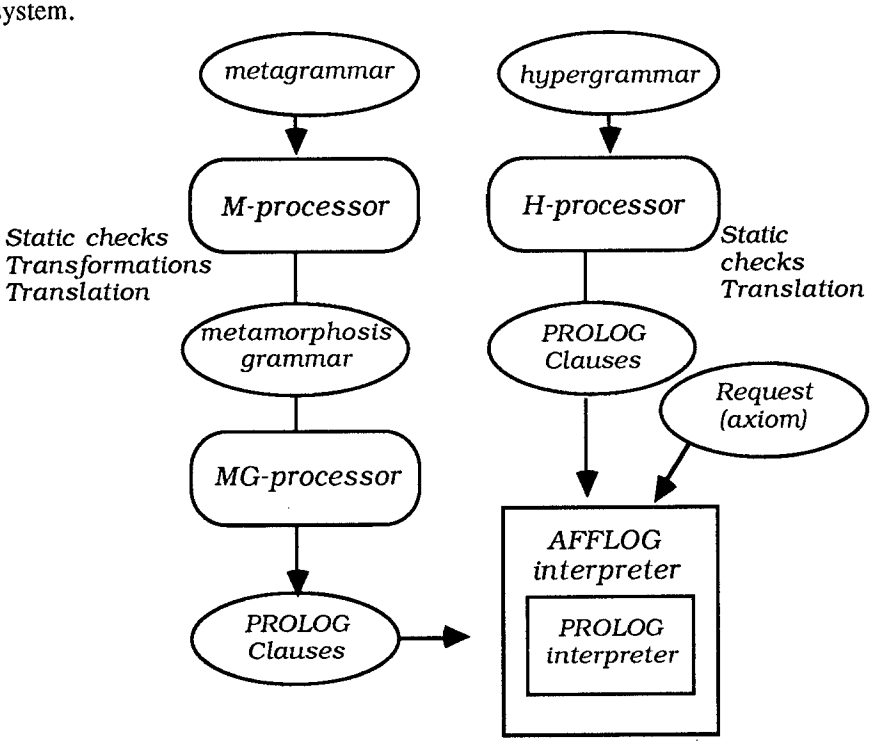

Fig. 2 : An overview of the AFFLOG two-level grammar processor

The M-processor compiles the metagrammar when it is well-formed (see § 2.2.3). The H-processor compiles the hypergrammar provided that the static analysis is successful (see § 2.3).

Let us introduce AFFLOG semantics thanks to a self-explanatory program. We consider the question of an environment construction in a small programming language.

E1 (comments in italic style)

The metagrammar is :

```
DECS  <=  DEC  DECS ; EMPTY.
          one DECS  is a  list of  DEC which may be reduced to EMPTY
DEC   <=  IDF  TYPE.
          one DEC is an IDF  followed by a TYPE
TYPE  <=  "ref"  TYPE ; BASIC.
BASIC <=  "int" ; "bool".
          a BASIC is the string ""int"  or the string "bool"
IDF   <=  "i" ; "j" ; "k" .
```

The hypergrammar is :

```
declarations (DEC DECS)   <- declaration (DEC), more_declarations (DECS).
declaration (IDF TYPE )   <- ?IDF, type (TYPE) .
more_declarations (DECS)  <- ?";", declarations (DECS).
type (ref TYPE)           <- ?"ref" , type (TYPE).
type (BASIC)              <- ?BASIC.
```

The axiom is S : declarations(DECS).

?X reads a symbol (PROLOG term) on input and unifies it with X.

Given the *query* : <- declarations (DECS1).
If the input string is "k bool ; j ref ref int" then the provided answer is declarations(k bool j ref ref int), a variable-free instance of the axiom S.

Let us introduce more precisely AFFLOG semantics.

2.2 AFFLOG Semantics

An AFFLOG text can be considered either as an extended affix grammar or as a logic program. We will define its semantics from both points of view.

Formal definition of AFFLOG grammars

An AFFLOG grammar is a 8-uple $(M_n,M_t,M_r,H_n,H_t,H_r,T,S)$

Here, (M_n,M_t,M_r) is called a *metagrammar* where M_n is a finite set of *non-terminal affixes*, M_t is a finite set of *terminal affixes* and M_r is a finite set of context-free *affix rules* i.e. a subset of $M_n \otimes (M_n \cup M_t)^*$ and M_n and M_t are assumed to be disjoint.

A variable is an element of M_n possibly concatenated to a natural number. If V_m is the set of variables which are built on the nonterminal m, V is the union of the sets V_m, $m \in M_n$. For all m in M_n, the language generated by the CF-grammar (M_n,M_t,M_r,m) defines the domain for the variables of V_m.

Then, Hg=(H_n,H_t,H_r,T,S) is called an *hypergrammar.*

H_n is a set of (basic) *hypernotions.* These hypernotions are built with a functor name and parameters which are *grammatical terms* (sentential forms of a CF-grammar (M_n,M_t,MR,m)). We use the following syntax :

functor_name (e_1,...,e_n) where $\exists m_i \in M_n$, $m_i \Rightarrow^* e_i$ (by application of affix rules)

H_t is a set of *predicates.* The right-hand side of a predicate definition contains only PROLOG predicate calls and thus can not generate a language.

H_r is a finite subset of $H_n \otimes (H_n \cup H_t \cup T)^*$ called the set of *hyperrules.*

$T=TT \cup TM$ is the *terminal alphabet* (whose symbols are prefixed by '?') where TT is a finite set of terminal symbols which appears in the right-hand side of the hyperrules while TM is a finite set of terminal symbols which belong to the domains of the variables (terminal affixes which are terminal symbols). For the above example E1, $T=\{";","ref"\} \cup L(IDF) \cup L(BASIC)$.

$S \in H_n$ is the *axiom.*

The *uniform replacement rule* is the basic mechanism for language production in a two-level grammar. Following [Maluszynski-84], an hyperreplacement is any string homomorphism θ on $(V \cup M_t)^*$ which replaces variables occurring in the grammati-cal terms by legal productions according to the metarules.

The images of hypernotions under hyperreplacements are called hypernotion instances. A ground instance is called a protonotion (a variable-free instance of an hypernotion). The set of protonotions is the nonterminal alphabet of a CF-grammar which generates a language : the *protogrammar.* Considering the extension of hyperreplacements to hyperrules (hyperrules instances), the set of ground hyperrules is the possibly infinite set of (context-free) rules of the protogrammar. Ground instances of the axiom constitute the set of axioms of the protogrammar.

L(G), the *language* generated by an AFFLOG grammar G is a set of terminal strings whose elements are derived from ground instances of S (using protorules).

In fact, we work with partially instanciated hypernotions and hyperrules and construct a stepwise refinement of hyperrules in order to find derivations in G. Thus, the problem of *grammatical unification* must be solved : how can we compute an hyperreplacement θ such that $\theta(h)=\theta(h')$ where h and h' are hypernotions ?

When it exists, θ is called a grammatical unifier for h and h' and is given as a set of substitutions. The grammatical unification problem is a kind of string equation problem where the domains of variables are context-free languages [Maluszynski-82b]. The AFFLOG grammatical unification procedure is presented in § 2.2.3.

Let us consider a second example, E2, whose axiom is instr (ENV,TYPE). This AFFLOG program assigns a type to a simple assignment instruction given an environment produced by E1.

E2	*Context-free definition*	*Static semantics*
INSTR ::	ID , ":=" , EXPR.	int ⊗ int -> int & bool ⊗ bool -> bool
EXPR ::	PRIM , "+", EXPR ;	int ⊗ int -> int
	"(", EXPR, "=", EXPR,")";	bool ⊗ bool -> bool & int ⊗ int -> bool
	PRIM.	bool -> bool & int -> int
PRIM ::	ID ; NBR.	

Metagrammar
ENV <= ITEM ENV ; EMPTY.
ITEM <= ID TYPE.
TYPE <= 'int' ; 'bool' ; 'ref' TYPE .
PRIM <= ID ; NBR.
ID <= "i"; "j"; "k";....
NBR <= "1"; "2"; "3";....

Hypergrammar
(h1) instr (ENV, TYPE) <-
?ID, ?':=', type (ID,ENV,TYPE), expr (ENV,TYPE).
(h2) expr (ENV, 'int') <-
?PRIM , ?'+' , type (PRIM,ENV,'int'), expr (ENV,'int').
(h3) expr (ENV,TYPE) <- ?PRIM , type (PRIM,ENV,TYPE) .
(h4) expr (ENV, 'bool') <-
?'(', expr (ENV,TYPE) , ?'=' , expr (ENV,TYPE) , ?')' .
(h5) type (NBR,ENV,'int').
(h6) type (ID, ID TYPE ENV, TYPE) .
(h7) type (ID, ID1 TYPE1 ENV, TYPE) <-
dif (ID,ID1), type (ID,ENV,TYPE).
Axiom
S : instr (ENV,TYPE).

Query : <- type (i int k int, j bool,TYPE)
If the input string is "j := (i +1 = k)", we get TYPE=bool.

Grammatical semantics

The *specified relation* of a grammar G whose axiom S is the hypernotion $A(t_1,...,t_n)$ is the set of S instances which can produce some terminal strings of L(G) : that is $\exists\omega\in T^*$ such that $A(t'_1,...,t'_n)\Rightarrow^*\omega$ where t'_i is an instance of t_i.

Example :
instr (i int j int, int) belongs to the specified relation for E2 (ω = "i:=j").

A term t'_i may contain variables in which case $A(t'_1,...,t'_n)$ describes a relation where t'_i would not be variable-free. Hence, these terms define a subset of the language generated by the metagrammar. For example, instr (ID int ID1 int, int) describes the set of tuples associated to the generic assignment "ID := ID1 ".

The *characteristic relation* of G is a binary relation which links the specified relation to the generated sentences ($\omega\in L(G)$). The couple <instr(i int j int, int), "i:=j"> is an example of tuples over the characteristic relation defined by E2.

Thus, one may associate a subset of the specified relation to each sentence of L(G). String ω is called the *control*. It allows language-controlled computations of a specified relation subset.

Example : noun ("mal","sing") <- ?"bob"

Arguments "mal" and "sing" are the properties of the noun "bob". But "bob" is used as a control string which selects the instance noun("mal",sing") among the possible instances of noun(tp,nb) (tp∈ {"mal","fem"}, nb∈ {"sing","plur "}).

Operational semantics

The operational semantics of an AFFLOG program is described by the procedure which computes instances of the axiom S for a given goal or query i.e. which computes a subset of the characteristic relation. Let us introduce some auxiliary concepts and discuss the problem of grammatical unification in AFFLOG.

We write $\theta_1(N)$ to denote that a grammatical unifier θ_1 is applied to the hypernotion N ; $\theta_1 \circ \theta_2$ denotes the composition of θ_1 and θ_2 ; MGGU denotes the most general grammatical unifier for two grammatical terms t_1 and t_2. If θ is the MGGU of t_1 and t_2, then θ is given as a set of substitution pairs [x/y].

$N =_g N'$ means that $N=A(t_1,...,t_n)$ and $N'=A(t'_1,...,t'_n)$ can be grammatically unified. We write that $N \in {}_gNs$, $N \in H_n$, $Ns=(H_n)^+$ (N grammatically belongs to the set Ns) if at least one hypernotion $N' \in Ns$ such that $N =_g N'$ exists.

Unification in AFFLOG

Grammatical unification is the basic mechanism for context transmission. In AFFLOG, it works on the syntactic structures of the grammatical terms i.e. trees whose nodes are labelled by nonterminal affixes and whose leaves are terminal affixes. These representations are abstractions of grammatical terms presented as terms of an initial algebra associated to the metagrammar. By considering a CF-grammar as a specification whose equation set is empty, the initial algebra operators are defined by the production rules of the CF-grammar [Goguen-77] and the mapping between strings and algebraic terms is performed by classical parsing technics [Saidi-92a].

Hence, we must check that for each grammatical term, there exists a CF-grammar (M_n,M_t,M_r,m) which generates it. The unification of two terms produces a set of pairs [x/y] where x is a variable whose associated nonterminal symbol in the metagrammar derives y. The efficiency of the unification algorithm can be improved by user-defined modes associated to the hypernotions. The following notations are used : '+' for ground, '-' for free and '?' for any (e.g. for E2 instr(+,-), expr (+,-)...).

Given the following (meta)grammar and two grammatical terms T_1 and T_2

```
INSTR <= ID, ":=", EXPR.
EXPR <= PRIM, "+",  EXPR ; "(", EXPR, "=",  EXPR, ")"; PRIM.
PRIM <= ID ; NBR.

T1 :    k := (PRIM = EXPR)
T2 :    ID := (x = j + 1)

Tree(T1)= instr(id("k"), ":=", expr("(", expr(prim(PRIM)), "=", expr(EXPR), ")"))
Tree(T2)= instr(id(ID), ":=" , expr ( "(", expr(prim(id("x"))),
                          "=" , expr(prim(id("j"))), "+", expr(prim(nbr("1"))), ")" ) )
```

We must perform the following confrontations : ID v. k, id("x") v. PRIM and EXPR v. expr(prim(id("j"))), "+", expr(prim(nbr("1")))). These are terms of the algebra associated to the CF-grammar.

A function from these terms onto $(T \cup V)^*$ gives the following MGGU :
θ=[ID/k, PRIM/x, EXPR/ j+1]

In order to compute unifiers, the metagrammar must not contain ε-rule and must not be "infinitely ambiguous" to enable our non-deterministic parser to find a finite set of parse trees [Saidi-92ab]. A well-formed metagrammar is compiled into definite clauses and the programmer gets a parser for it.

Resolution in AFFLOG

The AFFLOG operational device uses the SLD-resolution whose unification part is extended to grammatical unification and a PROLOG-like strategy ($\mathcal{R}$) (considering hypernotions as predicates and hyperrules as definite clauses).

A query consists of (B,δ) where B is the goal and δ a control string.

For a program P and a goal B such that $S =_g B$, a SLD-derivation of $P \cup \{B\}$ following $\mathcal{R}$ consists of a sequence B, B_1 ... of goals, a sequence H_1, H_2 ... of hyperrules from P and a sequence θ, θ_1... of MGGU such that B_{i+1} can be derived from B_i and H_{i+1} by using θ_{i+1} according to $\mathcal{R}$.

The first unification between axiom S and goal B (the question) gives $\theta(B)=\theta(S)$ where θ is the MGGU of B and S. The resulting goal is $(\theta(S),\delta)$ with $\delta \in T^*$.

Let $\delta=\omega\alpha$ where $\omega,\alpha \in T^*$, $\omega=\omega_1...\omega_k$ is the string already parsed while $\alpha=\alpha_1...\alpha_p$ is the string to be parsed.

If at step i, we have $B_i=\theta_i(A)$ (A <- $A_1,...,A_m$. $A_j \in (H_n \cup H_t)$) and the control string $\omega\alpha$, then we can produce $B_{i+1}=\theta_{i+1}(A')$ and the control string $\omega'\alpha'$ from B_i by using the MGGU θ_{i+1} via $\mathcal{R}$ by one of the following rules :

① If A_1=?X, X is a variable, if X and α_1 can be grammatically unified and $[X/\alpha_1]$ is the resulting unifier
then $\theta_{i+1}=\theta_i o[X/\alpha_1]$, $\omega'=\omega\alpha_1$, $\alpha'=\alpha_2...\alpha_p$, $A'=A_2,...,A_m$.

② If A_1=%q, and if the PROLOG predicate q succeeds (the empty string is derived) producing θ_{i+1} as resulting unifier
then $\omega'=\omega$, $\alpha'=\alpha$, $A'=A_2,...,A_m$.

③ If $A_1 \in H_n$ and if H <- $H_1,...,H_q$. is the hyperrule chosen following $\mathcal{R}$ and if θ_{i+1} is the MGGU of A_1 and H
then $\theta_{i+1}(A_1)=\theta_{i+1}(H)$
$\omega'=\omega$, $\alpha'=\alpha$, $A'=H_1,...,H_q,A_2,....,A_m$.

2.3 Checking and debugging AFFLOG programs

Construction and use of a Pattern Grammar

A parse in an AFFLOG grammar G is directed by a context-free parse of the input using a *pattern grammar* G_S such that $L(G_S) \supseteq L(G)$ [Maluszynski-84]. Derivation trees in G_S could be used when trying to build derivations in the AFFLOG program. Note that this process may not terminate.

Let us call *definition hypernotion* (resp. *application hypernotion*) an hypernotion which occurs on the left-hand side (resp. right-hand side) of the hyperrules. Let H_D be the set of definitions and H_A be the set of applications. Clearly $S \in H_A$. Let $\equiv$ be the binary relation on H (subset of H_n) defined as follows :

$h \equiv h'$ if $h =_g h'$ and $h \in H_D$ and $h' \in H_A$ or $h \in H_A$ and $h' \in H_D$.

This relation is called the cross-reference relation and its transitive closure is an equivalence relation on H. We denote by [h] the equivalence class of h.

In order to produce the pattern grammar $G_S=(N_S,T,R_S,S_S)$ we proceed as follows : the set N_S of the equivalence classes [h] is computed ; T is the terminal alphabet ; S_S is the equivalence class [S] and in each hyperrule $r \in H_r$, we replace each hypernotion $h \in H_n$ by [h]. Predicate symbols for which the generated language is empty are then removed from N_S and metarules associated to the variables which occur on the right-hand side of the hyperrules (prefixed by '?') are added to R_S. Finally, G_S is translated into a metamorphosis grammar [Colmerauer-75].

For E2, we produce the following grammar :

```
A       ::   ID, ':=' , C , B.
B       ::   PRIM, '+', C, B.
B       ::   PRIM, C.
B       ::   '(', B, '=', B, ')'.
C       ::   C ; .
ID      ::   ...
NBR     ::   ...
PRIM    ::   ...

where   A = {instr (ENV, TYPE)}
        B = {expr (ENV,TYPE), expr (ENV,'int'), expr (ENV,'bool')}
        C = {type (ID,ENV,TYPE), type (ID, ID TYPE ENV, TYPE)...}
```

$S \in_g A$

Note that before looking for a derivation which might produce an instance of S, the input string could be parsed according to G_S by bottom-up parsing methods [Nilsson-86].

A special case of AFFLOG program must be reported : if $T=\emptyset$ and $S \Rightarrow^* \varepsilon$ we have a PROLOG program for which there is no user-defined language control. Grammatical unification enables typed PROLOG programming and in that case every instance of the specified relation can be computed.

E3 : List concatenation
L <= EMPTY ; X L.
X <= symbol.
conc (EMPTY,L,L) .
conc (X L,L1, X L2) <- conc (L,L1,L2).
S=conc (L1,L2,L3).
The pattern grammar is : A .
A :: A .
with A={ conc(EMPTY,L,L), conc(X.L1,L2,X.L3), conc(L1,L2,L3) }, $S \in_g A$.

Pattern grammar construction provides a grammar whose language is a super set of the one generated by the original AFFLOG grammar. Experimentations might be done to check if a given string belongs to the language generated by the pattern grammar. Some special conditions are required to get a one-to-one correspondence between the derivations in both grammars [Wegner-80]. Another use of the pattern grammar is to enable an off-line derivation. One may extract all dependencies between grammatical terms and build an equational system to be solved when a derivation is constructed in the pattern grammar [Maluszynski-82a, Isakowitz-91].

A first check on the hypergrammar consists in considering an application hypernotion (h) and looking for some definition hypernotion (h') which is grammatically unifiable to h. We check that the axiom is grammatically unifiable with the query before the activation of the derivation procedure. If user-defined modes are associated to the hypernotions, the above checks take them into account.

Construction and use of a Hypernotion schematas

Being given the equivalence classes, an algorithm computes a *most specific generalizer* (MSG) for each of them. This algorithm is a kind of anti-unification algorithm [Plotkin-69]. Informally, the MSG for a set of hypernotions is an hypernotion such that all the elements of the set are particular instances.

For E2, we get MSG(A) = instr (ENV, TYPE)
MSG(B) = expr (ENV,TYPE)
MSG(C) = type (ID,ENV,TYPE)

In [Saidi-92b], we propose an algorithm which computes the MSG and guaranties that for each equivalence class, the MSG is unique under some conditions. We also check whether there exists an unique derivation for each member of a given equivalence class (starting from the associated MSG).

This helps the translation of AFFLOG programs into DCGs and may provide the transparency condition to EAG (see bellow).

Grammatical consequence and use of Hypernotion schematas

For an AFFLOG program whose pattern grammar is $G_S=(N_S,R_S,T,S_S)$, the following sets are built recursively :

$L_0=T$

$L_{n+1}=L_n \cup \{ A \,/\, (A :: W) \in R_S,\ W=w_1,...,w_k\ ,\ k \geq 0,\ w_i \in (N_S \cup T), w_i \in L_n \}$

So, $\forall n$, if $A \in L_n$ with $n>0$ then $A \Rightarrow^* x$, $x \in T^*$

The algorithm stops at step k if it does not add any new element to L_{k-1}. Thus, k is such that $L_k=L_{k-1}$, while for $n<k$, $L_{n-1}\neq L_n$ and L_n contains at least one more symbol (a nonterminal $\in N_S$) than L_{n-1}. Since N_S is finite, clearly k is finite and we have $k \leq \text{card}(N_S)$.

$X \in N_S$ is called a *grammatical consequence* of P if $X \in L_k$. This concept is naturally extended to every hypernotion occurring in P.

For E2, we get (see equivalence classes A,B and C previously given) :
L_0 = { ID, NBR, PRIM, ":=", "+", "=", "(", ")"}
$L_1 = L_0 \cup \{C\}$
$L_2 = L_1 \cup \{B\}$
$L_3 = L_2 \cup \{A\}$
k=3 , $S_S \in L_3$.

For E3 we get : $L_0=\emptyset$, $L_1=L_0 \cup \{A\}$, k=1, $S_S \in L_1$

The grammatical consequence concept is used to perform some static checks : the set $L=L_k-L_0$ is computed and then used to build a metarule whose right-hand side is the disjunction of MSGs associated with the equivalence classes represented by L elements. This metarule (called ATOM in [Maluszynski-84]) is very close to the "PREDICATES" definitions section in Turbo Prolog [Borland-86]. If the programmer specifies it, the consistency between the specified rule and the computed one is checked : the right-hand side elements of the specified rule must belong to the computed equivalence classes.

For E2, having L={A,B,C}, we get :
ATOM <= instr (ENV, TYPE) ; expr (ENV,TYPE) ; type (ID,ENV,TYPE).

The construction of the ATOM metarule could provide a transparency condition [Maluszynski-84] for AFFLOG grammars provided that the following open problem is solved :

If there is a unique derivation for every equivalence class elements starting from the corresponding MSG then the metagrammar whose axiom is ATOM is not ambiguous.

Static analysis is finished by checking whether all hypernotions can be derived from ATOM and whether for each application there exists at least one definition which can be grammatically unified with it. The hypergrammar is then translated into PROLOG clauses.

From AFFLOG grammars to Definite Clause Grammars

AFFLOG grammars can be translated to DCGs if we succeed in computing the MSGs. In this case, grammatical terms are replaced by their termal representations which are parse trees. MSGs guide this translation since they provide informations about the derivation tree roots, that is, functional symbols are known when rewriting strings into compound (PROLOG) terms.

The following DCG is generated for E2 :

```
(r1)    instr(env(ENV), type(TYPE))        -->
                [ID], [':='], type(prim(id(ID)),env(ENV),type(TYPE)),
                expr(env(ENV),type(TYPE)), {id(ID)}.
```

```
(r2)    expr(env(ENV), type(ent))          -->
                [PRIM] , ['+'] , type(prim(id(PRIM)),env(ENV),type(int)) ,
                expr(env(ENV),type(int)) , {prim(PRIM)}.

(r3)    expr(env(ENV), type(ent))          -->
                [PRIM] , ['+'] , type(prim(nbr(PRIM)),env(ENV),type(int)) ,
                expr(env(ENV),type(int)) , {prim(PRIM)}.

(r4)    expr(env(ENV), type(TYPE))         -->
                [PRIM], type(prim(id(PRIM)),env(ENV),type(TYPE)),
                {prim(PRIM)}.

(r5)    expr(env(ENV), type(TYPE))         -->
                [PRIM] , type(prim(nbr(PRIM)),env(ENV),type(TYPE)) ,
                {prim(PRIM)} .

(r6)    expr(env(ENV), type(bool))         -->
                ['('] , expr(env(ENV), type(TYPE)) , ['='] ,
                expr(env(ENV), type(TYPE)) , [')'] .

(r7)    type(prim(nbr(NBR)), env(ENV), type(int)) --> [].

(r8)    type(prim(id(ID)), env([item(id(ID), type(TYPE)), env(ENV)]),
                        type(TYPE))     --> [] .

(r9)    type(prim(id(ID)), env([item(id(ID1), type(TYPE1)), env(ENV)]),
                        type(TYPE))     -->
                {dif(prim(id(ID)), prim(id(ID1)))} ,
                type(prim(id(ID)), env(ENV), type(TYPE)).

id(X)           :- X=i ; X=j ; X=k.
dif(X,Y)        :- X \=Y.
prim(X)         :- id(X) ; nbr(X).
nbr(X)          :- X=1 ; X=2 ; X=3 .
```

Note that rules (r1)...(r5) take into account the fact that the H-schemata for *type* (i.e. type(PRIM, ENV, TYPE)) represents all the possible configurations of parse trees associated to ID and NBR. These trees have PRIM at their roots and either a number (NBR) or an identifier (ID) as their leaves.

Query example :
If the input string is "j:=(i+1=k)" and if the value of ENV is <i int k int j bool>, the query instr (ENV, TYPE) is submitted to PROLOG as the following term :

```
?- instr(env([item(id(i),type('int')),env([item(id('k'),type('int')),
                        env([item(id('j'),type('bool')),env('empty')])])]),
                type(TYPE),[j, :=, '(',i,'+',1,'=',k,')'], []).
```

The computed value for TYPE is given by type (TYPE)=type (bool) meaning that TYPE value is "bool".

3 Conclusion

We presented some of our current work on AFFLOG programs static analysis. The AFFLOG interpreter has been implemented in PROLOG (2500 lines). A grammatical unification algorithm as well as a grammatical resolution procedure have been developed. The resolution can be controlled by terminal string derivations belonging to the language specified by the program. We emphasize the static checks, not only those which are related to type-sensitive string unification but also to hypergrammar checking.

AFFLOG has been designed to study program construction by means of two-level grammars (in particular the extended affix grammars). It supports specification debugging within a Wide Spectrum Grammatical Programming Framework [Boulicaut-92]. These specifications can be used (and transformed) when we want to produce translators using the STARLET/GL compiler compiler [Beney-90].

Apart from this interpreter, we have also implemented various logic grammar processors : Definite Clause Grammars, Metamorphosis Grammars, Definite Clause Translation Grammars [Abramson-84] and Gapping Grammars [Dahl-84]. This toolbox is used to develop experimental systems in order to study two-level grammar properties from both the theoretical and the practical points of view. For instance, partial specification of modes and automatic error processing are actually studied.

References

Abramson-84 ABRAMSON (H.). Definite Clause Grammars and the Logical Specification of Data Types as Unambiguous CF- Grammars. Proc. of Int. Conf. of FGCS, Tokyo (J), 1984, p. 678-685.

Beney-90 BENEY (J.), BOULICAUT (J.F.). STARLET : an affix-based compiler compiler designed as a Logic Programming System. In : Proceedings of the Third International Workshop on Compiler Compilers CC'90, October 22-24, 1990, Schwerin (G), D. HAMMER Ed., Springer-Verlag, LNCS 477, p. 71-85.

Borland-86 BORLAND Int. Turbo Prolog Owner's Handbook, Scott Valley (USA), 1986.

Boulicaut-92 BOULICAUT (J.F.). Towards a Wide Spectrum Grammatical Programming Framework. Ph-D Thesis: INSA Lyon, february 1992, 320 p. (in Frencl)

Colmerauer-75 COLMERAUER (A.). Metamorphosis Grammars. Research Report GIA (Aix-Marseille II, Luminy), november 1975.

Dahl-84 DAHL(V.), ABRAMSON(H.). On Gapping Grammars. in : Proceedings of the 2nd. ICLP, Uppsala (S), 1984.

Deransart-88 DERANSART (P.), MALUSZYNSKI (J.). A Grammatical View of Logic Programming. in : Proceedings of the 1st Int. Workshop PLILP 88, Orléans, mai 1988, (P. DERANSART, B. LOHRO, J. MALUSZYNSKI Eds), Springer-Verlag, LNCS 348, p. 219-251.

Goguen-77 GOGUEN (J.A.) & al. Initial Algebra Semantics and Continuous Algebras. JACM Vol. 24, n°1, january 1977, pp. 68-95.

Hehner-83 HEHNER (E.C.R.), SILVERBERG (B.A.). Programming with Grammars : An Exercise in Methodology-Directed Language Design. The Computer Journal, Vol.26, 1983, p. 277-281.

Isakowitz-91 ISAKOWITZ (T.). Can we transforme logic programms into attribute grammars ? RAIRO Informatique Théorique et Applications. Vol. 25, n° 6,1991, p. 499-543.

Kowalski-79 KOWALSKI (R.). Logic for Problem Soving. North-Holland, 1979.

Maluszynski-82a MALUSZYNSKI (J.), NILSSON(J.F). A comparison of the logic programming language Prolog with two-level grammars. in : Proceedings of the 1st ICLP, Marseille (F), 1982, M. van CANEGHEM Ed., p. 193-199.

Maluszynski-82b MALUSZYNSKI (J.), NILSSON (J.F.). Grammatical Unification. Information Processing Letters, Vol.15, n°4, 1982, p.150-158.

Maluszynski-82c MALUSZYNSKI (J.) , NILSSON(J.F). A version of Prolog based on the notion of two-level grammars. Working paper at the "Prolog programming environments workshop", Linköping Institute of Technology (S), march 1982, 15 p.

Maluszynski-84 MALUSZYNSKI (J.). Towards a programming language based on the notion of two-level grammars. TCS, Vol.28, 1984, p.13-43.

Näslund-87 NASLUND (T.). An experimental implementation of a compiler for two-level grammars. in : Proceedings of the 2nd int. symp. on methodologies for intelligent systems, (Z.W.RAS & M. ZEMANKOVA Eds.), pp.424-431.

Nilsson-86 NILSSON (U.). AID : an Alternative Implementation of Definite Clause Grammars. New Generation Computing, 1986, p. 385-398.

Pereira-80 PEREIRA (F.C.N.), WARREN (D.H.D.). Definite Clause Grammars for Language Analysis : a survey of the formalism and a comparison with ATN. Artificial Intelligence, Vol 13, 1980, p. 231-278.

Plotkin-69 PLOTKIN(G.) A Note on Inductive Generalization. Machine intelligence 5, 1969, p.153-163.

Saidi-92a SAIDI (S.). Associative Unification in the AFFLOG Grammatical Logic Programming System. in : Proceedings of JFPL'92, 25-27 may 1992, Lille (F), (J.P Delahaye Ed.), p.107-126 (in french)

Saidi-92b SAIDI (S.). Grammatical Extensions to Logic Programming. Ph-D Thesis : Ecole Centrale de Lyon, may 1992, 172 p. (in french).

Torii-84 TORII (K.), MORISAWA (Y.), SUGIYAMA (Y.), KASAMI (T.). Functional programming and logical programming for the telegram analysis problem. in : Proceedings of 1st IEEE Int. Symp. on Logic Programming, Atlantic City (USA),1984, p. 463-472.

Watt-74 WATT (D.A.). Analysis-oriented two-level grammars. Ph.D. thesis : University of Glasgow, 1974, 285 p.

Wegner-80 WEGNER (L.M.). On Parsing Two-Level Grammars Acta Informatica,Vol.14, 1980, p. 175-193.

Ecully, june 5th, 1992