Reversibility in a Constraint and Type based Logic Grammar: Application to Secondary Predication

Palmira Marrafa
FLL - Universidade de Lisboa / ILTEC
Ava Engo Arantes e Oliveira, 40 (Lte 41) , 8 E.
P- 1900 LISBOA
Portugal

Patrick Saint-Dizier
IRIT Université Paul Sabatier
118, route de Narbonne
F-31062 TOULOUSE cedex France
e-mail: stdizier@irit.irit.fr

Abstract

In this document, we present a formalism for natural language processing which associates type construction principles to constraint logic programming. We show that it provides more uniform, expressive and efficient tools for parsing and generating language. Next, we present two abstract machines which enable us to design, in a symmetric way, a parser and a generator from that formalism. This abstract machinery is then exemplified by a detailed study of secondary predication within the framework of a principled-based description of language: Government and Binding theory.

Introduction

Lexical as well as grammatical and discursive knowledge required to understand or to produce natural language utterances is usually a description which is independent of the sentence production or comprehension 'algorithms'. It comes naturally into mind to have a common, shared knowledge base of what language is, independently of its potential uses. Besides well-known advantages of uniformity and transparency, this point of view is very convenient for the computer scientist who does not have to integrate into his parsers or generators the unavoidable updatings required by grammar development. The linguistic knowledge is thus specified in a declarative way in different modules (lexical, grammatical, discursive, ...) and different strategies are applied which refer to these data (directly for interpreters or via the production of a compiled code for compilers). This approach can however be realized more or less easily depending on the formalisms used to describe language phenomena.

In this document we introduce new advanced tools of the Logic Programming framework and show that they contribute to meeting the requirements imposed by the manipulation and the control of large amounts of data required by both the parsing and the generation procedure. We first consider logical types which are a declarative and easy-to-use tool and formalism which permit a grammar writer to encode knowledge in a very flexible and principled-based way.

In addition to types, we introduce new active constraints of the Constraint Logic Programming (CLP) framework which allow us to treat and to check for consistency of constraints throughout the whole generation procedure and not to only evaluate them when they are given in the programme or grammar. These active constraints are fully declarative and can be used by any type of parsing/generation process.

CLP introduces a greater expressive power together with a higher efficiency since the resolution of constraints is postponed till they can be properly evaluated and since constraints have to be always true and consistent with each other. Finally, a feature of active constraints is that they are usually independent of the way they are processed, they are thus strategy independent and can equivalently be used for parsing and for generation.

To make reversibility concrete in our system, we develop in this document two different abstract machines based on type construction and constraint satisfaction which give the foundations of a parser and a generator using the same source of declarative linguistic knowledge. The differences between these machines exemplifies the 'technical' differences one may have between parsing and generation processes.

1. A type based description language

Three major types of operations are at the basis of the typed-based language we have designed for language processing, namely:

- the expression of type construction to generate phrase structures,
- the expression of dependencies (either local or long-distance) between types,
- the expression of well-formedness constraints on types.

Types here refers to the usual data structures in computer science. We now informally present the syntax of our type-based language. It is directly derived from the syntax of Login (Aït-Kaçi and Nasr 86). The syntactic representation of a structured term is called a ψ-term. It consists of:

(1) a *root symbol*, which is a type constructor and denotes a class of entities,

(2) *attribute labels*, which are record field symbols. Each attribute denotes a function in extenso, from the root to the attribute value. The attribute value can itself be a reference to a type.

(3) *coreference constraints* among paths of labels, which indicate that the corresponding attributes denote the same function. They are indicated by variables. Here is an example:

```
person( id => name(first => string,
                   last => X: string),
        born => date(day => integer,
          month => monthname,
                     year => integer),
        father => person( id =>
              name(last => X ))).
```

The root symbol is *person; id, born* and *father* are three sub-ψ-terms which have either constants or types as values. *X* indicates a coreference. All different type structures are tagged by different symbols. Notice also that in the latter field only relevant information about person is mentioned. Infinite structures can also be specified by coreference links. Variables are in capital letters, constants in small letters.

2. Dealing with constraints

We have extended the type description framework with active constraints and have given them a Constraint Logic Programming (Colmerauer 90, Jaffar and Lassez 87) interpretation, permitting us to have a more principled-based description of language. The general form of a type is :

`Type :- Constraints.`

We view constraints as part of the type:

`(Type :- Constraints)`

is itself a type, subsumed by Type.

The simplest constraint is the precedence constraint:

`precede(X,Y),`

where X and Y are of type string. This constraint imposes that the string X precedes of the string Y. When processing a sentence, precedence constraints on constituents are stated in the grammar rules and possibly at the lexical level. At each stage i of the processing, there is a partial order P1(i) on the words and structures already processed. At the end of the process, precedence constraints give all the possible word orderings which meet the constraints of the grammar. In the case of parsing, constraints imposed by the order of words in the input string must be coherent with the precedence results of the parse.

The next constraint imposes the presence of an attribute in a type:

`has(Attribute, Type)`

where Attribute is either an attribute label or a full pair attribute-value and Type is a reference to a given type. This constraint imposes that at some stage there is an attribute in Type which is subsumed by or equal to Attribute. Informally, (1) when incoherence with Attribute is detected or (2) when Type is fully constructed, the non-satisfaction of has(Attribute,Type) will provoque backtracking. This constraint permits us to encode thematic role assignment and focus management, and also to encode the inclusion of a set of values into another.

The last class of constraint is mainly related to the expression of long-distance relations between sentence constituents. Within the framework of types, the notion of long-distance is somewhat obsolete since there is no ordering relation on subtypes in a type (attributes may be written in any order). Thus, the notion of long-distance dependency will be here formulated as a sub-type co-occurence constraint. This constraint emerged from Dislog (Saint-Dizier 87, 89). Very briefly, the co-occurence of two or more subtypes in a larger type is expressed by the constraint: pending(A,B) where A is a type specification and B is a list of type specifications. Informally, this constraint means that

A originates the pending of the sub-types in B, in other terms that A can be used if, somewhere else in the main type (corresponding for example to a full sentence), all the sub-types in B are used with identical substitutions applied to identical variables.

3. Processing Language with types and constraints

We will mainly present here simple, motivational examples. A more abstract syntactic description will be given in section 6 which will more fully motivate our approach. The examples given in this text show that our description language can accomodate principled-based descriptions of language like Government and Binding theory as well as lexically and head driven descriptions like in the HPSG framework.

In the following simple examples, we only have two main type constructors:

- x0 corresponding to lexical entries,
- xp corresponding to phrase structures.

Here is the description of the lexical entry corresponding to the verb *to give*:

```
x0( cat => v,  string => [give] ) :-
  pending(x0(cat => v),
 [xp( cat => n, string => S1, role => patient ),
  xp( cat => p,  string => S2,
    role => recipient) ] ),
  precede([give],S1),  precede(S1, S2).
```

This entry says that *give* is a verb which subcategorizes for an np with role patient and a pp with role recipient; np and pp are left pending. The string S1 generated from the np has to precede the string S2 generated from the pp. These constraints will be treated at the level of the type describing the structure of a vp. *The whole description x0 construction and related constraints is the type of the verb to give*. Let us now consider the construction of a vp with an np and a pp complements. To the construction of a vp type corresponds the generation of a (set of) string(s) corresponding to a vp, this is stored in S. We then have the following construction:

```
    xp( cat => v,  string => S,
          const1 => x0(cat => v ),
          const2 => X : xp(cat => n),
          const3 => Y : xp( cat => p)  ) :-
              has(role, X), has(case, X),
              has(role, Y), has(case, Y).
```

The constraints has(role,X) and has(role,Y) impose that the constituents const2 and const3 have a role assigned at some level of the type construction process. The same situation holds for case. This is a simple expression, for example, of the case filter in GB theory. Notice that most pending situations are satisfied locally, which limits complexity.

4. An abstract machine for type construction in a parsing process

Parsing a sentence is constructing a well-formed type describing the sentence structure. We present in this section an abstract machine which describes how types are constructed. This machine is based on the procedural semantics of Prolog but it resembles a push-down tree automaton whose stack is updated each time a subtype is modified.

There are two kinds of type constructors: those corresponding to non-terminal structures (such as xp and x1 in our examples) and those corresponding to terminal structures (e.g. x0). We now present a step in the construction of a type. It can be decomposed into 3 levels:

(1) current state σ_i :

$c0(a_1 => t_1,\ a_2 => t_2,\ \dots,\ a_n => t_n)$,

(2) selection in the current programme P of a type construction specification:

$c1(b_1 => t'_1, \dots, b_m => t'_m)$

such that t_1 subsumes it or unifies with it modulo the mgu θ_i.

(3) New state σ_{i+1} : t_1 is replaced by :

$c1(b_1 => t'_1, \dots, b_m => t'_m)$,

with, as a result, the following type:

$c0(a_1 => c1(b_1 => t'_1, \dots, b_m => t'_m),\ a_2 => t_2,\ \dots,\ a_n => t_n)\ \theta_i$

The process goes on and processes t'_1. The type construction strategy is here similar to Prolog's strategy and computation rule : depth-first and from left to right. The main difference at this level with SLD-resolution is that only types corresponding to non-terminal structures are expanded. Informally, when a type t_j corresponds to a terminal structure, an attempt is made to find a terminal type description t'_j in the programme which is subsumed by or unifies with t_j and, if so, a replacement occurs. t'_j is said to be in a *final state*. If t'_j does not exist, backtracking occurs. The next type description immediately to the right of t'_j is then treated in the same manner. The type construction process successfully ends when all subtypes corresponding to terminal symbols are in a final state and it fails if a terminal type description t_p cannot reach a final state. The initial state is :

```
xp( cat => sentence,
     string => [ string,to,parse] ).
```

4.2. Extension of the abstract machine to constraints

The above abstract machine can be extended in a simple way to deal with constraints. Constraint resolution mechanisms are similar to usual constraint logic programming systems like Prolog III. The three above levels become:

(1) current state σ_i represented by the couple:

$< c0(a_1 => t_1, a_2 => t_2, \ldots, a_n => t_n), S >$

where S is the set of current constraints,

(2) selection in the current programme P of a type construction specification:

$c1(b_1 => t'_1, \ldots, b_m => t'_m) :- R.$

where R is the set of constraints associated to c1, and t_1 subsumes or unifies with t'_1 modulo the mgu θ_i.

(3) New state σ_{i+1} characterized by the following couple:

$< c0(a_1 => c1(b_1 => t'_1, \ldots, b_m => t'_m),$
$a_2 => t_2, \ldots, a_n => t_n) \theta_i ,$
$S \cup R \cup subsume(t_1,c1(b_1 => t'_1, \ldots,$
$b_m => t'_m)) >$

with the condition that the new set of constraints must be satisfiable with respect to the constraint resolution axioms defined for each type of constraint and, if not, a backtracking occurs. At this level constraints simplifications may also occur.

The output of the parsing process may be simply a syntactic tree, but it may also be a logical formula, similar to the one used and presented in section 5. We however think that both processes, parsing and generating, need not necessarily respectively produce and start from the same abstract internal representation.

5. An Abstract Machine for Language Generation

From the above declarative descriptions of language construction, an abstract machine for language generation can also be defined. At the level of type construction, generation proceeds by monotone increasing restrictions: a phrase structure is described by a type constructor linking a set of subtypes. This operation introduces a restriction on the possible left and right contexts that each of the subtypes could potentially have if they were independent from each other. The degree of generality of the selected type constructor linking those subtypes can be subject to various interpretations. Finally, generation is guided by the semantic representation from which a sentence is uttered. As shall be seen, the semantic representation will determine the computation rule and the subgoal selection procedure. It is thus much more deterministic than its parsing process counterpart.

Let us now briefly consider the abstract machine for language generation. The general technique, that we have already exemplified in (Saint-Dizier 89), consists in:

- (1) writing a formal grammar of the semantic representation from which the generation process starts,
- (2) identifying the phrasal units and the lexical units (and intermediate units if necessary) which can be associated to the symbols of that formal grammar,
- (3) associating generation points to these symbols (terminal and non-terminal) which will generate natural language fragments based on a consultation of the grammatical and the lexical system (these generation points could be added automatically).

For example, if the formal grammar of the semantic representation of quantified noun phrases is:

```
Quant_np --> det([Quant, Var], Np,
                        Rest_of_sentence).
Np --> and( Noun, Modifiers ).
```

We then have, for example, and informally, the following generation points, where the call *p(formula, string, corresponding syntactic category)* is used to process the semantic representation:

```
p(det([Quant,Var],Np,Rest_of_sent),
        Type ) :-
     p(Quant, Type1),   p(Np, Type2),
   generation_point(Type1, Type2, Type3),
     p(Rest_of_sentence, Type4),
   generation_point(Type3, Type4, Type).
```

```
p(and(Np,Mod),Type) :-
      p(Np, Type1),   p(Mod,Type2),
     generation_point(Type1, Type2, Type).
```

The relation between a predicate (or an argument) and a word is established by a call to a lexical entry as follows:

```
p(Predicate, Type) :-
        Type,    has(Type,
           sem_rept =>    Predicate ).
```

Informally, Type1 and Type2 are constructed from the treatment of the quantifier and the noun phrase, they are then combined, in the first rule above, by means of the first call to generation_point, resulting in Type3. This generation point includes the treament of the string of words being generated (including the precedence constraints on the words generated from lexical insertion) and the treatment of more abstract features such as category, inflection or semantic characteristics. Finally, the second call to *generation_point* integrates Type3 with Type4, the latter being the type associated to the remainder of the sentence. The result is Type.

Generation points support by themselves the generation strategy. A model of these generation points is given below by means of an abstract machine. As can be noticed, calls to generation points occur after the parse of the corresponding semantic structure. This means that calls to generation points will be stacked (by Prolog) and will be then unstacked in the reverse order they have been stacked: the strategy is then *bottom-up*.

Generation points determine, by means of a call to the grammatical system, the resulting syntactic category and the way the partial strings of words in Type1, Type2 and Type4 are assembled. The way types are constructed by generation points is modelled by the following abstract machine. At this level, we generalize the generation points to take into account any number of subtypes, and not only two as shown in the examples.We claim that this method is general and can be used from most current semantic representations (such as, for example, DRT or Conceptual Graphs).

The abstract machine for language generation can be described by its initial state and a step in the construction procedure. It has the general form of a finite state tree automaton. The initial state is σ_0, it is the empty type. Let us now consider a step σ_i .

1. Two cases arise: it is either

(a) a set of subtypes from which a more general type can be constructed :

σ_i = (a) $(C_1, C_2, \ldots, C_n)$ is an unordered sequence of subtypes ; or

(b) it is a single type : σ_i = D1

2. Type constructor selection:

(a) let DC be such that: DC has exactly k attributes $const_j$, $k \leq n$,

and DC is of the form:

$DC := xp(\ldots, const_1 \Rightarrow C'_1, \ldots, const_k \Rightarrow C'_k)$

and :

for all $j \in [1,k]$, subsume(C'_j, C_j)

(notice that the C_j are not necessarily the jth element of the list given in 1 above, notice also that the type constructor DC contains the subtypes $const_q$ together with other information like category and morphology.)

or **(b)** D' (single type)

3. σ_{i+1} = **(a)** $(DC , C_{k+1}, \ldots, C_n)$

for all $i, j \in [1,k]$

or **(b)** (D1, D').

The type constructor DC contains the subtypes $const_q$ together with other information like category and morphology. It should be noticed that the constructor DC is selected according to a subsumption criterion, which is more general and powerful than standard unification. It better corresponds to the process of incremental generation of phrases. The process ends when a type with category *sentence* is reached. This is a terminal state in the automaton, more precisely it is the root of the tree automaton, since our generation system proceeds in a bottom-up fashion.

Let us now consider the step 2 above devoted to the selection of a type constructor. This selection is mainly guided by the generation points given in the formal grammar of the semantic representation. They indeed select between cases (a) or (b) and in case (a) they directly characterize which of the C_i will be included in the type construction at the current stage. Finally, since active constraints associated to type descriptions can be executed at any time, the constraint resolution mechanisms which maintain constraint coherence are independent of the generation strategy. In other terms, these mechanisms are independent of the way and the order constraints are added to the set of active constraints.

The abstract machine which handles types and constraints is the following. It is represented by a tuple: <type, set of active constraints>.

We then have:

1. σ_i =

(a) $< (C_1, C_2, ..., C_n), S >$ sequence of subtypes C_i and of active constraints S

(b) < D1, S >

2. Type constructor selection:

(a) < DC, R> where R is the set of constraints associated to DC and such that:

i) same restrictions as above on DC and

ii) R is consistent with S

(b) < D' , R > (single type) with R consistent with S.

3. σ_{i+1} =

(a) $< (DC , C_{k+1}, ..., C_n), (S \cup R \cup (subsume(C'_j \Rightarrow C_j)) \text{ for all } j \in [1,k]) >$

(b) $< (D1, D'), S \cup R >$

At the end of the generation process, the set of possible admissible surface sentences can be directly derived from the precedence constraints which may not be a total order (some words may have different positions).

6. An Application to Secondary Predication

We now present a more elaborate and comprehensive example which will further motivate our approach. Secondary predication is described at both lexical and syntactic levels, the intertwining of several constraints makes it simpler to describe in a fully declarative way. The description is thus independent of its use: parsing or generation. This gives a good application example of the specification and use of our formalism and system for a real phenomenon of much importance to natural language system designers.

6.1 A linguistic approach

Secondary Predication is a term used in the literature to denote a very productive structural relationship in many languages: the relationship between a subject and a predicate, the subject being assigned a thematic role by that predicate and by an obligatory thematic role assigner in the sentence, namely the verb. For instance, in (1)

(1) Mary drinks the water cold

the *water*, the direct object of *drinks*, is assigned a thematic role by this verb and another one by the adjective *cold*. Then, *water* is, at the same time, an object for *drinks* and a subject for *cold*. In other terms, *water* integrates - as an object - a primary predication which corresponds to the whole sentence, and - as a subject - a secondary predication which corresponds to the sequence *the water cold*.

6.1.1 Object-oriented Predicates

Secondary predication is not an uniform or an homogeneous phenomenon, neither from the point of view of a specific language, nor from a crosslinguistic one. We will describe here some of the most relevant structural properties and lexical constraints of this type of construction in French. Let us begin by considering the French sentence corresponding to (1):

(2) Marie boit l'eau froide.

(2) is an ambiguous sentence as can be illustrated by the paraphrases below (the English translations of the examples are, all of them, literal translations):

(2) (a)(Marie boit l'eau qui est froide ("Mary drinks the water which is cold")

(b) Quand Marie boit l'eau, l'eau est froide. ("When Mary drinks the water, the water is cold")

Considering the interpretation in (2)(a), the adjective is part of the direct object of the verb, which is not the case for the interpretation in (2)(b). Then, *l'eau froide* can have the structure

(3)(a) [NP[NP l'eau] [AP froide]]

or the structure:

(3)(b) [NP_i l'eau][AP_i froide].

In (3)(a) *froide* is a modifier of *eau*, while in (3)(b) it behaves as a predicate, assigning a secondary thematic roleto the NP. The predication relationship is expressed by coindexation.

Let us now consider the sentence (4):

(4) Marie boit l'eau minérale

("Mary drinks the water mineral")

In spite of its superficial structural resemblance with the example above, (4) is not ambiguous, the interpretation corresponding to the paraphrase (b) being not available:

(4)(a) Marie boit l'eau qui est minérale

("Mary drinks the water that is mineral")

but :

*(4)(b) *Quand Marie boit l'eau, l'eau est minérale*

("When Mary drinks the water, the water is mineral")

This means that the possibility of having or not having an object-oriented secondary predication depends on the semantic nature of the adjective. Moreover, there also exist semantic co-occurrence restrictions between the adjectival predicate and the verb:

*(5) *Marie boit l'eau congelée*
("Mary drinks the water frozen")

(5) is excluded because something frozen cannot be drunk. Notice that the presence of an adjective in sentences like (2) is optional, in opposition to what happens in sentences like (6) (for the same interpretation of the verb):

(6) Marie considère l'eau froide
("Mary considers the water cold")
*(6)(a) *Marie considère l'eau*
"Mary considers the water")

What we can infer from the fact that (6)(a) is ruled out is that: (i) *considérer* (to consider) does not subcategorize for an NP, then *froide* can not be a modifier of *l'eau*; (ii) if *froide* is not a modifier of *l'eau* it must be a predicate, but, in this case, we dont have the structure presented in (3)(b). In fact, *l'eau froide* behaves like a clausal phrase. It can even be replaced by a completive sentence (the semantic interpretation remaining the same) as exemplified in (6)(b):

(6)(b) Marie considère que l'eau est froide
("Mary considers that the water is cold")

We have then empirical evidence to analyse *l'eau froide* in (6) as a clause, a "small clause" using an usual label in the literature (since the categorial status of the small clause is irrelevant for our purposes, we will only use the symbol "SC" to refer to this constituent, assuming the small clauses analysis proposed by Stowell (1981) and Stowell (1983)). As a consequence, *l'eau froide* is a predication having the structure in (7):

(7) [SC[NP$_i$ l'eau] [AP$_i$ froide]]

In this case it is the whole predication, and not only its subject, which is theta-marked by the verb. Stricto sensus, we have not a secondary predication, nevertheless, the contrastive analysis remains important since the two kinds of structures are superficially very similar.

As largely assumed in the GB framework (Chomsky (1981) and (1986)), predication is configurationnaly constrained: subject and predicate must be reciprocally m-commanded, that is, all maximal projections (phrase levels) dominating one of them must dominate the other one. Given this condition and the facts we have examined, (8) and (9) are appropriate representations (we use here X-bar notation only when relevant), respectively, for (2) and (6):

(8) [S [NP Marie] [V" [V'[V boit] [NP$_i$ l'eau]] [AP$_i$ froide]]]

(9) [S [NP Marie] [V"[V'[V considère] [SC [NP$_i$ l'eau] [AP$_i$ froide]]]]]

Although attached to different nodes inside V" (while in (8) the subject of the secondary predication occupies the direct object position and its predicate is in a weak adjunction position (in the sense of Demonte (1988)), in (9) subject and predicate are together in direct object position), the predications we have considered so far involve only adjacent elements. Let us now examine sentence (10):

(10) La lessive rend le linge blanc
(The washing makes the clothes white")

Similary to what happens in (6), the sentence is ruled out if the adjectif is not present:

*(10)(a) *La lessive rend le linge*
("The washing makes the linge")

With respect to these facts it seems to be natural that sentence (10) is structurally identical to (9). Nevertheless, (10)(b), which is equivalent to (10), does not support this hypothesis:

(10)(b) La lessive blanchit le linge
("The washing whitens the clothes")

As we can observe, *blanchit*, a verbal predicate, can replace the verb - *rend* - and the adjectival predicate - *blanc* (for the same semantic interpretation). Then, *rend blanc* behaves like a single predicate. At the same time, *blanc* is a secondary predicate for *linge*. Following a proposal by Marrafa (1983) and (1985) for similar cases in Portuguese, we consider *rend blanc* as a discontinuous complex predicate and we express the relationship between the two elements that constitue it by co-superscription. Therefore, (10) has the structure (10)(c),where k indicates the discontinuity in the predicate *rend-blanc :*

(10)(c) [S[NP La lessive] [V"[V'[V^k rend] [NP$_i$ le linge] [AP^k_i blanc]]]]

6.1.2 Subject-oriented predicates

Discontinuity can also be an obligatory property of a certain kind of secondary predication, namely in the case of subject-oriented predicates. (11) is an example:

(11) Jean dansait triste

("John dansed sad")

Since *Jean* is a proper noun, it can not be modified. Then *triste* is necessarily a predicate for John. Although the subject of *triste* is the main subject and not an NP in object position as in the above sentences, there are semantic co-occurrence constraints between the verb and the adjectival predicate, as illustrated below:

*(11)(a) *Jean dansait repenti*

("John dansed repented")

Taking into account these constraints and not violating the m-command condition refered to above, we represent the adjectival predicate as an strong adjunction (again in the terms of Demonte (1988)) to V", the syntactic representation of (11) being, then, (11)(b):

(11)(b) [S[NP$_i$ Jean] [V" [V" [V' [V dansait]]] [AP$_i$ triste]]]

Notice that continuous and discontinuous secondary predications can co-occurre in the same sentence:

(12) Jean$_i$ boit l'eau$_j$ froide$_j$ triste$_i$

("John drinks the water cold sad")

It is also interesting to point out that, in certain cases, sentences are ambiguous with respect to continuous and discontinuous secondary predication:

(13) Jean$_i$ laisse son amie$_j$ triste$_{i\ j}$

("John left his girlfriend sad")

To summarize, secondary predication can be associated to different types of structure and to continuous or discontinuous elements. Moreover, there are numerous and different semantic co-occurrence restrictions of different types affecting the lexical items involved.

6.2 An Implementation in terms of types and constraints

We now show how the above examples are expressed both at syntactic and lexical levels. The full syntactic structures are given under (8), (9), (10c) and (11b). The structure in (8) says that the AP is a sister of the V' (noted in the grammar as V with bar level 1) and that the AP is co-indexed with the object NP, the co-indexation relation is left pending since it is preceded by the V' description. We have the following construction:

```
xp( cat => v, string => SV,
    const1 => x1( cat => v, string => S3),
   const2 => xp( cat => a, string => S4,
           index => I ) :-
  pending(xp(cat => v),[x1( cat => v,
      string => T,  const1 => x0( cat => v,
      string => S1), const2 => xp( cat => n,
        string => S2, index => I ) ) ] ),
       precede(S1,S2), precede(S3,S4).
```

Since the AP is not obligatory (it is a weak adjunct), there is nothing said about it in the lexicon.

Construction (9) introduces a small clause (noted here as sc). Since it is not necessarily contiguous to the V', but only dominated by the V', we need a pending constraint. The type construction is the following:

```
x1( cat => v, string => SV,
    const1 => x0( cat => v, string => S1 ),
   const2 => xp( cat => sc, string => S2 ) )  :-
   pending(x1(cat => v),[x1( cat => sc,
    string => SV,  const1 => xp( cat => n,
      index => I, string => S3), const2 =>
    xp( cat => a, string => S4,index => I ) ) ] ),
     precede(S1,S2), precede(S3,S4).
```

The lexical entry of the verb (here *considerer*) has a pending constraint for the small clause: the verb subcategorizes for a small clause.

Construction (10c) introduces a double indexation but no long-distance dependency for the compound predicate 'rend-blanc'. We represent it as follows:

```
x1( cat => v, string => SV,
   const1 => x0( cat => v,
                    compound_pred => K,
                        string => S1),
   const2 => xp( cat => n, index => I,
                          string => S2 ),
   const3 => xp( cat => a, index => I,
    string => S3, compound_pred => K ) )  :-
      precede(S1,S2), precede(S2,S3).
```

Finally, the construction given in (11b) introduces a long-distance relation between an NP in subject position and an AP which is in object position. To handle this phenomenon, we have to go up to the sentence level, that we will represent here for simplicity as s (instead of, for example, COMP). The type construction is the following:

```
xp( cat => s, string => SV,
  const1 => xp( cat => n, string => S1,
                          index => I ),
   const2 => xp( cat => v, string => S2  ) ) :-
    pending(xp(cat => s),[xp( cat => v,
                string => T,
   const1 => xp( cat => v,string => S3),
   const2 => xp( cat => a,
                string => S4, index => I ) ) ] ),
          precede(S1,S2), precede(S3,S4).
```

At the lexical level, the adjoined AP is not mentioned, since it is not syntactically necessary (but it might be necessary from a semantic point of view, as also for case (8) above).

7. Specific features of our approach

Our approach can be contrasted mainly with the usual systems based on unification grammar (UG) formalisms (Shieber, 86), (Emele and Zajac 90). The first major difference is that the unification and rewriting mechanisms usually associated with UG are replaced by a more constraining operation, type construction, which always *proceeds by sucessive restrictions* (or monotone increasing specialisation) each time a type is further expanded. From that point of view, our approach also substantially differs from (Aït Kaçi and Nasr, 86) who propose a powerful and semantically clear mechanism for typed unification associated to type inheritance.

Next, we have *a single operation: type construction*; we do not have on the one hand grammar rules and on the other hand, associated to each rule, a set of equations to deal with feature values and constraints. The constraints we have associated with our types are not of the same nature and cannot be compared to the equations of UGs. They are moreover a part of the type.

Constraints added to types are interpreted within the *CLP framework*, this permits us to have a more expressive and powerful constraint system, which is also more efficient and simpler to write. Constraint satisfaction is not indeed guaranteed at the level they are given, but throughout the whole type construction process.

Our approach is compatible with the current principled-based approaches to describing languages. This is exemplified in section 4 by the constraints on role and case assignments. In a more general way, the description language we have presented here is particularly *appropriate for highly abstract descriptions of language*, which corresponds to several current trends in computational linguistics. Our description language is, in the same time, well-adapted to deal with lexical-based approaches to language processing (those approaches like lexicon grammars where the lexicon plays a central role) and to describe representations developed within lexical semantics.

Finally, a constraint like pending *generalises the notion of long-distance dependency to several other kinds of dependencies*. This generalization is in particular a consequence of the fact that type structures do not have any ordering on subtypes and they cannot, thus, directly express the difference between remote and close constituents.

Besides these general properties, our approach has several interesting properties which are more specific to reversibility. First, the common data shared by the two processes is all the linguistic data which is specified in a declarative way: lexical and grammatical. The semantic composition rules are the same. In the case of generation, they are translated into a parser of the formal grammar of this semantic representation. It should be pointed out that the parser given in section 5 can be generated automatically.

Both processes also have a lot of elements in common at the procedural level: the type construction mechanisms are identical, the major difference at this level being the selection rule, which is, in the case of generation, guided by the semantic form from which the process starts. The other difference is that parsing proceeds a priori top-down in the case we have exemplified (it could also proceed bottom-up). Generation proceeds bottom-up, for reasons explained in section 5. From this difference it results that the starting type in the case of parsing is a general type corresponding to sentence whereas there are no starting type in the case of generation, the starting points being the types corresponding to the predicates appearing in the logical formula, which are deduced from an operation close to lexical insertion. If the parsing process were bottom-up, then the starting types would be the same and the subsumption operation would also be used instead of the standard unification.

Finally, and most importantly, the constraint system that we have presented is fully independent of the strategies used and of the direction of the process: generation of parsing. This is a consequence of the fact that constraints are evaluted only when there is sufficient available information to evaluate them and also that their coherence with the other constraints is checked throughout the whole proof construction procedure. The variables which are used by active constraints are thus global variables.

Conclusion

In this document, we have first defined a formalism based on types and active constraints of the Logic Programming framework and have shown that it is well-appropriate to describe language constructions. We have in particular illustrated it by focussing on secondary predication, an important phenomenon in language processing. Finally, we have shown that our formalism is particularly appropriate to be used by a parser and by a generator, in a symmetric way, and we have defined for that prupose two abstract machines. This work is now fully implemented in Sicstus Prolog (which allows the writing of constraint resolution mechanism) on a Sun workstation. Since constraints are so far meta-interpreted, we cannot make real comparisons with existing NLP systems. A significant result is however the much smaller number of backtraking operations that we have observed.

Acknowledgements

We thank N. Hathout for his insightful comments on this work, which was supported by the PRC Communication Homme-Machine and the French ministry of research.

References

Aït-Kaçi, H., Nasr, R., LOGIN: A Logic Programming Language with Built-in Inheritance, *journal of Logic Programming*, vol. 3, pp 185-215, 1986.

Chomsky, N., *Lectures on Government and Binding*, Foris, 1981.

Chomsky, N., *Barriers*, Linguistic Inquiry monograph no 13, MIT Press, 1986.

Colmerauer, A., An Introduction to Prolog III, *CACM* 33-7, 1990.

Demonte, V., Remarks on Secondary Predicates: C-Command, Extraction and Reanalysis, *The Linguistic Review* 6, pp 1-39, 1988.

Emele, M., Zajac, R., Typed Unification Grammars, in proc. COLING'90, Helsinki, 1990.

Jaffar, J., Lassez, J.L., Constraint Logic Programming, *Proc. 14th ACM Symposium on Principles of Programming Languages*, 1987.

Marrafa, P., Teoria das Pequenas Oraçoes vs Teoria da Predicaçao: Controlo, Critério Tematico e Principio de Projecçao, ms, FLL-University of Lisbon, 1983.

Marrafa, P., A Construçao Transitiva-Predicativa em Português, FLL-University of Lisbon, 1985.

Saint-Dizier, P., Contextual Discontinuous Grammars, 2nd NLULP, Vancouver 1987 and in: *Natural Language Understanding and Logic Programming II*, V. Dahl and P. Saint-Dizier Edts, North Holland, 1988.

Saint-Dizier, P., A generation Strategy based on GB Principles, *proc. 2nd European workshop on language generation*, Edinburgh, 1989.

Saint-Dizier, P., Constrained Logic Programming for Natural Language Processing, *proc. ACL-89*, Manchester, 1989.

Sheiber, S., An Introduction to Unification-Based Approaches to Grammar, *CSLI lecture notes no 4*, Chicago University Press, 1986.

Stowell, T., Origins of Phrase Structure, PhD. dissertation, MIT, 1981.

Stowell, T., Subject across Categories, *The Linguistic Review* 2, pp 285-312, 1983.