# Processing Language with Logical Types and Active Constraints

Patrick SAINT-DIZIER
IRIT Université Paul Sabatier 118, route de Narbonne
31062 Toulouse cedex FRANCE
e-mail: stdizier@irit.irit.fr

## ABSTRACT

In this document, we present a language which associates type construction principles to constraint logic programming. We show that it is very appropriate for language processing, providing more uniform, expressive and efficient tools and treatments. We introduce three kinds of constraints, that we exemplify by motivational examples. Finally, we give the procedural semantics of our language, combining type construction with SLD-resolution.

## Introduction

With the development of highly parameterized syntactic theories like Government and Binding theory and Head-Driven phrase structure grammars and with the development of theories where rewriting and unification plays a central role, like Categorial grammars and Unification Grammars, there is an increasing need for more appropriate and more efficient feature systems.

Feature systems must be designed to preserve the adequacy, the expressiveness and the explanatory power of the linguistic system that one wants to model. Real parsing as well as generation systems often require the manipulation of large sets of features, these systems must therefore offer a great flexibility in the specification of features in grammar symbols and a significant modularity so that each linguistic aspect (morphological, categorial, ...) can be dealt with independently. Features are often subject to various constraints. These constraints cannot always be evaluated at the level they are formulated (e.g. a feature value is not yet known) but have to be evaluated later and must be true throughout the whole parsing or generation process.

The development of principled-based approaches to language processing also require the definition of more abstract formal systems to handle in an adequate way these principles. Principles indeed often apply not at grammar rule level but they involve a large part of a parse tree. They must be expressed by a constraint system which is global to the whole grammar and not local to a rule, as for example, in DCGs.

These brief considerations have motivated our approach: syntactic rules are viewed as type constructions on which constraints are applied. These constraints are themselves part of the type. To give an appropriate expressive power to constraints and an efficient interpretation, they are interpreted within the Constraint Logic Programming framework.

In the next sections, we introduce our description language based on types and constraints. We then give motivational examples which clarify its use. Then we give its procedural interpretation and a constraint resolution mechanism system.

## 1. A typed-based description language

Three main 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 describe phrase structures,
- the expression of relations (either local or long-distance) between types,
- the expression of well-formedness constraints on types.

The term type refers here to structured data representation. They must not be confused with types defined in linguistics, as in the categorial system.

The starting point of our description language is CIL (Mukai 85), a language designed to model Situation Semantics which permits the expression of some constraints on typed descriptions called complex indeterminates; and Login (Aït-Kaçi and Nasr 86), a typed-based language with a built-in inheritance schema. To these languages we have added specific feature treatments and constraints usually necessary for language processing that we now find in advanced unification grammar systems (Sheiber 87, Emele & Zajac 90). We have also provided a specific declarative and procedural semantics merging type construction and resolution of constraints, viewed as active constraints of the constraint logic programming framework (noted hereafter as CLP) (Colmerauer 90), (Jaffar and Lassez 87).

We now informally present the syntax of our type-based language. It is directly derived from the syntax of Login. 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 or to an instance of a type.

(3) *coreference constraints* among paths of labels, 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. To this description, we have added the treatment of negation on constants, and the treatment of conjunctions and disjunctions, in a way quite similar to (Johnson 90). Taxonomic relations between identifiers are also taken into account in the unification mechanism. These features will not however be developed, since this is not central to what we want to explain here.

This formalism permits us to define type inheritance and the possibility to define in a clean way classes and subclasses corresponding to structured linguistic objects (Saint-Dizier 91).

## 2. Dealing with constraints

The general form of a type is :

```
Type :- Constraints.
```

Constraints belong to the following classes:

- constraints on attribute values not yet known,
- constraints on values of two attributes,
- constraints on the existence of an attribute possibly with a given associated value,
- constraints on the co-existence of attributes (to express dependencies),
- constraints expressing precedence relations on strings of words.

The first two classes of constraints being developed by F. Günthner (Günthner 88) within the framework of Prolog III, we will here concentrate on the three last types of constraints, which are quite different in nature from the two first ones. We view constraints as part of the type: (Type :- Constraints) is itself a type, subsumed by Type.

The linear precedence constraint:

```
precede(X,Y),
```

where X and Y are of type string. It imposes that the string X precedes of the string Y. Precedence constraints on constituents are stated in the grammar rules and at the lexical level. At each stage i of a parse, a partial and coherent order P1(i) on the words and structures already processed can be constructed. On the other hand, the input sentence to parse has a strict order P2 on words. Then, at each stage of the parsing process, P1(i) and P2 must be satisfiable. As shown in the ID/LP framework, having precedence relations permits us to have a more general and flexible description of phrase structures. The CLP interpretation of precedence permits us to have a more efficient system because backtracking will occur as soon as a precedence violation is detected.

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

```
has(Attribute,Type)
```

where Attribute is either an attribute label or a pair attribute-value (a sub-ψ-term) and Type is a reference to a 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 also permits us 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), that we now briefly present.

A **Dislog clause** is a finite, unordered set of Horn clauses $f_i$ of the form:

$$\{f_1 , f_2 , \ldots\ldots , f_n \}.$$

The informal meaning of a Dislog clause is: *if a clause $f_i$ in a Dislog clause is used to construct a given proof tree, then all the other $f_j$ of that Dislog clause must be used to construct that proof tree, with the same substitutions applied to identical variables.* Moreover, there are no hypothesis made on the location of these clauses in the proof tree. For example, the following Dislog clause composed of two Prolog facts:

*{ arc(a,b), arc(e,f) }.*

means that, in a graph, the use of *arc(a,b)* to construct a proof is conditional to the use of *arc(e,f)*.

If one is looking for paths in a graph, this means that all path going through the *arc(a,b)* will have to go through the *arc(e,f)*, or conversely.

A Dislog clause thus permits us to express co-occurence of clauses in a proof tree. The constraint stating that all identical variables in an instance of a Dislog clause must be substituted for the same terms permits the transfer of argument values between non-contiguous elements in a very convenient way. A Dislog clause can be subject to various types of restrictions such as: linear precedence constraints on the $f_i$ , modalities on applications of some $f_i$ and the specification of bounding domains in which an Dislog clause instance must be fully used (Saint-Dizier 89).

The co-occurence of two 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 types in B, in other terms that A can be used as a type constructor if, somewhere else in the main type (corresponding to sentence), all the types in B are also used as type constructors with identical substitutions applied to identical variables. Notice that this constraint is not equivalent to a conjunction of has(X,T) constraints because the has(X,T) constraint imposes that T is fully defined whereas pending(A,B) does not impose, a priori, any constraints on the location in the main type of the types in B. The constraint resolution mechanism of this constraint is given in section 6.

## 3. Parsing with types and constraints

We first present simple, motivational examples. A more abstract syntactic description of the X-bar system follows and shows more clearly the expressive power of the formalism. The following examples 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 (which also follow principles, but not in the same sense).

### 3.1 A simple grammatical system:

In the following 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,
   role => patient, case => acc ),
   xp( cat => p, role => recipient,
                  case => dative ) ] ).
```

This entry indicates that *give* is a verb which subcategorizes for an np with role patient and case accusative and a pp with role recipient and case oblique, which are left pending since they do not necessarily immediately follow the verb in a sentence. These constraints will be dealt with by the type describing the structure of a vp. *The whole description x0 construction and the 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:

```
xp( cat => v,  string => S,
 const1 => x0(cat => v, string => S1 ),
 const2 => X : xp(cat => n, string => S2),
 const3 => Y : xp( cat => p, string => S3)) :-
       has(role, X), has(case, X),
       has(role, Y), has(case, Y),
       precede(S1,S2), precede(S2,S3).
```

The $const_i$ attributes in the type constructor *xp* permits the satisfaction of the pending constraints specified in the lexical entries. We view phrase structure type constructors both as a descriptive and a computational mean to construct structures. The constraints has(role,X) and has(role,Y) impose that the constituents const2 and const3 have a role assigned at some level in the type construction process. The same situation holds for case. This is a simple expression, for example, of the Case Filter and the θ-criterion in GB theory. Notice that most pending situations are satisfied locally, which limits complexity. Finally, notice that the denotation of this type is the set of sentences S which can be constructed and which meet the constraints.

### 3.2 Expressing X-bar syntax

Our description language permits the expression of most current versions of X-bar theory that we now illustrate. X-bar syntax is a typical example of type construction. Let us consider the rule:

$X^1 \dashrightarrow X^0$, complement.

The element $X^0$ is a direct reference to the type constructor x0, as described in the preceding section. We now show how $X^1$ is defined by the type constructor x1; the nature of the complement is induced from lexical descriptions given in x0:

```
x1(  cat => C, bar => 1,  string => S,
  head =>x0( cat => C, bar => 0, string => S1,
       complement => Z : xp( cat => Compl,
       bar => B1, role => R,  satisfied => 1) ),
  complement => xp( syntax => Z ,
       case => Case,   string => S2 ) ) :-
          atom(R), atom(Ca),
          precede(S1,S2),
          C =/= infl, C=/= comp,
          assign(C, Case).
```

Notice how a co-reference link is established by means of the variable Z between the subcategorization frame given in the head and the syntactic characteristics of the complement. The subcategorization data is not left pending since it is contiguous to the head and can be moved only by another mean, namely, within GB theory, by move-$\alpha$. The subcategorization of a complement is satified (i.e. satisfied => 1), the complement is assigned the appropriate $\theta$-role. Case is also assigned whenever appropriate, by the call to *assign_case*.

Similar rules can be defined for $X^2$ and adjuncts, with the difference that adjuncts are usually not obligatory. Rules involving non lexical categories like INFL and COMP are described in the same manner. However, at the level of INFL, the assignment of the external $\theta$-role and case to the subject position is carried out using a long-distance constraint, expressed in Dislog.

## 3.3 On Government

The notion of Government in GB theory introduces an interesting use of the constraint has to control the well-formedness of a construction. We now present a model for Government, so that $\theta$-roles and cases can be properly assigned. In what follows, we refer to Government and to the notion of Barriers as defined in (Chomsky 86).

Government can be modeled as a well-formedness constraint on $X^2$ categories. Indeed, each $X^2$ which can be a barrier (i.e. all categories except INFL) prevents a category above it from governing a category dominated by that $X^2$. Thus, for all rules of the general form:

$Z \rightarrow W, X^2, T.$

where Z, W and T are any kind of non-terminal symbol, a control has to be made on the well-formedness of $X^2$ if $X^2$ is a barrier. This control consists in three constraints:

- every N2 is assigned a case (Case Filter in GB),
- every N2 is assigned a thematic role ($\theta$-criterion),
- all obligatory subcategorization has to be satisfied (Projection Principle).

The two first constraints have been already given in 3.1, the latter is carried out by checking that the following sub-$\psi$-term is not present in the type constructor x2 corresponding to the category $X^2$:

```
  xp( obligatory => 1, satisfied => 0 )
```

which can be expressed by the negated constraint:

```
not(has(xp( obligatory => 1,
            satisfied => 0 ),Type).
```

The attribute *obligatory* comes from lexical description where it is specified whether an complement is obligatory or not. The attribute *satisfied* is properly instanciated to 1 when a complement is constructed (see 3.2).

## 3.4 On Long-Distance Dependencies

Let us finally consider an example of the expression of long-distance dependencies for which we use the pending constraint: wh-movement. Refering to X-bar syntax, the general annotated surface form is:

$[_{COMP}$ $PRO_i$ ........ $[_{N2}$ $trace_i$ ] ..... ]     as in:
$[_{COMP}$ $THAT_i$ John met $[_{N2}$ $trace_i$ ] yesterday ]

Within the framework of our type-based approach, a pending constraint specifies the co-occurence of two type constructions, which must be both used during the type construction process associated to the sentence being parsed. In our example, the first subtype constructor will describe the adjunction of an N2 to a COMP node (here COMP0) while the second subtype constructor will express that that N2 is constructed from a trace. A shared variable, I, represents the co-indexation link:

```
  { xp( cat => X:  comp0,   string => S,
    const1 => xp(cat => n, form => pro,
                     index => I, string => S1 ),
    const2 =>  xp( cat => X , string => S2 )   ,
   xp( cat => n,  form => trace,  string => S3,
      index => I )  }   :-    precede(S,S3).
```

Since the adjunction to COMP is always to the left of the trace, this Dislog clause can be translated into a single type specification by means of the pending constraint:

```
 xp( cat => X:  comp0,   string => S,
    const1 => xp(cat => n, form => pro,
                     index => I, string => S1 ),
    const2 => xp( cat => X , string => S2 ))  :-
   pending(xp(cat => comp0), xp( cat => n,
form => trace,  string => S3, index => I ) ),
   precede(S,S3).
```

To summarize, in lexical entries we express the subcategorization requirements and the precedence relations; in types expressing syntactic constructions, we have controls on the contents of types and pending constraints due to long-distance dependencies between sentence constituents.

## 4. An abstract machine for type construction

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 $\sigma_i$ :

$c_0( a_1 \Rightarrow t_1,\ a_2 \Rightarrow t_2,\ \ldots,\ a_n \Rightarrow t_n),$

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

$c_1(b_1 \Rightarrow t'_1, \ldots, b_m \Rightarrow t'_m)$

such that t1 subsumes it (or unifies with it) modulo the mgu $\theta_i$.

(3) New state $\sigma_{i+1}$ : $t_1$ is replaced by :

$c_1(b_1 \Rightarrow t'_1, \ldots, b_m \Rightarrow t'_m)$,

with, as a result, the following type:

$c_0(a_1 \Rightarrow c_1(b_1 \Rightarrow t'_1, \ldots, b_m \Rightarrow t'_m),$
$a_2 \Rightarrow t_2, \ldots, a_n \Rightarrow t_n)\, \theta_i$

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.

## 5. Extension of the abstract machine to handle 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 $\sigma_i$ represented by the couple:

$< c_0(a_1 \Rightarrow t_1, a_2 \Rightarrow t_2, \ldots, a_n \Rightarrow t_n), S >$

where S is the set of current constraints,

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

$c_1(b_1 \Rightarrow t'_1, \ldots, b_m \Rightarrow t'_m)$ :- R. where R is the set of constraints associated to $c_1$, and t1 subsumes (or unifies with) $t'_1$.

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

$< c_0(a_1 \Rightarrow c_1(b_1 \Rightarrow t'_1, \ldots, b_m \Rightarrow t'_m),$
$a_2 \Rightarrow t_2, \ldots, a_n \Rightarrow t_n),$
$S \cup R \cup \text{subsume}(t_1, c_1(b_1 \Rightarrow t'_1, \ldots,$
$b_m \Rightarrow t'_m)) >$

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. Mgu $\theta_i$ is replaced here by the subsumption constraint.

## 6. A Constraint Logic Programming interpretation of the 'pending' constraint

The pending constraint is interpreted within the Constraint Logic programming framework (Colmerauer 90, Jaffar and Lassez 87) in a very simple way. The constraint solving approach of the CLP corresponds better to programming practice and to programming language design. Constraints directly state properties on objects of the domain being dealt with which have to be always true, in contrast to coding them by means of terms. The CLP framework offers a global rule-based framework to handle and to reason about constraints.

The domain of objects on which constraints of a CLP interpretation of Dislog operate is a domain of types. Let us first consider a simple translation example of a Dislog clause into a CLP clause. A Dislog clause like:

{ a, b }

where a and b are type construction (TC) specifications, is translated as follows in CLP:

```
a :- pending(a, [b]).
b :- pending(b, [a]).
```

The constraint pending(A,B) states here that the TC A is at the origin of the pending TC B. The general case is interpreted as follows. Let us consider the Dislog clause:

{ A, B, ..., N }.

it is translated into a set of CLP clauses as follows:

```
A :- pending(A, [B, ..., N] ).
B :- pending(B, [A, ..., N] ).
...
N :- pending(N, [A, B, ...] ).
```

The constraint resolution procedure associated to pending consists in a simplification rule for the elimination of pending TCs when the co-occurence contraint is satisfied. This simplification rule is written as follows for the simple example given above in section 2:

pending(A,B)∧ pending(B,A) --> ∅ .

Notice that we have a symmetric treatment for A and B. The general simplification rule is the following, where LA, LB and LC are lists of pending TCs:

```
(pending(A, LA), pending(B, LB) -->
pending(A, LC) ) :-
  mb(A, LB), mb(B, LA),
  withdraw(B, LA, LC).
```

LC is the resulting pending list equal to LA minus B.

This constraint resolution mechanism can be further extended quite straightforwardly to handle linear precedence restrictions and modalities. Linear precedence constraints are dealt with independently from each other. The Dislog clause:

{ A, B, ..., X, ..., Y, ..., N } ..., X < Y , ...
is translated into a CLP clause as follows:

```
X <> pending(X, [A, B, ..., ..., Y, ..., N]) ∧ not
(pending(Y,[A, B, ..., X, ..., ..., N])).
```

The coherence control is the following:

```
pending(X, LA)∧not(pending(X, LA)) --> failure.
```

the simplification rule is:

```
not (pending(Y, [A, B, ..., X, ..., ..., N] )) --> ∅
```

or, more simply, since all negations are withdrawn at each stage: not (pending(_,_) --> ∅.

## 7. Specific features of our approach

Our approach can be contrasted mainly with the usual systems based on unification grammar (UG) formalisms. 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 substancially differs from (Emele & Zajac 90) 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.

The abstract machine we have described gives a clear procedural semantics to the system. A similar machine can be defined for natural language generation. Our description language has now being fully implemented in Prolog on a SUN workstation following the abstract machine description given above. The first version is an interpreter; a compiler is under development. Experiments with GB theory descriptions (Saint-Dizier 90) have been successfully carried out. It is however important to notice that our formalism is not specifically designed for GB theory and that it can express with the same accuracy other approaches such as HPSGs and lexicon grammars.

## Acknowledgements

This work owes much to discussions with Nabil Hathout.This work was supported by the French GRECO-PRC Communication Homme-Machine.

## 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., *Barriers*, Linguistic Inquiry monograph nb. 13, MIT Press 1986.

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

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

Günthner, F., Features and Values, Research Report Univ of Tübingen, SNS 88-40, 1988.

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

Johnson, M., Expressing Disjunctive and Negative Feature Constraints with Classical First-Order Logic, proc. ACL'90, Pittsburgh, 1990.

Mukai, K., CIL: Complex Indeterminate Language, *Fifth Generation Computing journal*, 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., Constrained Logic Programming for Natural Language Processing, *proc. ACL-89*, Manchester, 1989.

Saint-Dizier, P., Modelling Move-α and Government by a typed-based approach, GB-Parsing workshop, Geneva 1990.

Saint-Dizier, P., Condamines, A., An Intelligent Environment for the Acquisition of Lexical Data, proc. ACH/ALLC conference, Tempe AZ, 1991.

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