An Abductive Mechanism for Natural Language Processing Based on Lambek Calculus*

Antonio Frias Delgado[1] and Jose Antonio Jimenez Millan[2]

[1] Departamento de Filosofia
[2] Escuela Superior de Ingenieria de Cadiz
Universidad de Cadiz, Spain
{antonio.frias,joseantonio.jimenez}@uca.es

Abstract. We present an abductive mechanism that works as a robust parser in realistic tasks of Natural Language Processing involving incomplete information in the *lexicon*, whether it lacks lexical items or the items are partially and/or wrongly tagged. The abductive mechanism is based on an algorithm for automated deduction in Lambek Calculus for Categorial Grammar. Most relevant features, from the Artificial Intelligence point of view, lie in the ability for handling incomplete information input, and for increasing and reorganizing automatically lexical data from large scale *corpora*.

1 Introduction

1.1 Logic and Natural Language Processing

Natural Language Processing (NLP) is an interdisciplinary field where lots of research communities meet. Out of all NLP objectives, parsing is among the basic tasks on which other treatments of natural language can be founded. Development of efficient and robust parsing methods is a pressing need for computational linguistics; some of these methods are also relevant to Logic in AI whether they are founded on Logic or they use AI characteristic techniques.

Lambek Calculus (LC) for Categorial Grammar (CG) is a good candidate for developing parsing techniques in a logic framework. Some of the major advantages of CG lie in: (a) its ability for treating incomplete subphrases; (b) it is (weakly) equivalent to context free grammars, but (c) CG is radically lexicalist, it owns no (production) rule except logical ones; therefore, (d) syntactic revisions are reduced to type reassignments of lexical data of a given *lexicon*.

On the other hand, the Gentzen-style sequent formulation of LC for CG also presents several attractive features: (a) a well-known logical behaviour —LC corresponds to intuitionistic non-commutative multiplicative linear logic with non empty antecedent; (b) the cut-rule elimination, and hence the subformula property that is desirable with regard to its implementation.

* Partially supported by grant no. PB98-0590 of the Comisión Interministerial de Ciencia y Tecnología. We would like to thank two anonymous referees for their valuable comments.

M. Ojeda-Aciego et al. (Eds.): JELIA 2000, LNAI 1919, pp. 87–101, 2000.

When it comes to using LC in realistic tasks of NLP, one must admit that LC has two possible disadvantages: (a) its complexity is unknown; (b) in so far as it is equivalent to context free grammars, LC cannot account for several linguistic phaenomena. These limitations accepted, we encounter another kind of dificulties: the realistic tasks of NLP involve characteristic problems that cannot be solved by the sole use of deductive systems. A deduction is always something closed, in accordance with immovable rules; however our language understanding is robust enough and it succeeds even if partial information is lacking.

1.2 Learning and Revising Data

The AI researches intend to enlarge the logical machinery from the precise mathematical reasoning to the real situations in the real world. That means, for example: to learn from experience, to reorganize the knowledge, to operate even if the information is incomplete. The task of building robust parsers comes right into the goals of AI in a natural way.

The (informal) notion of robustness refers to the indifference of a system to a wide range of external disruptive factors [Ste92], [Men95]. Out of all desirable properties of a robust parser we focus on two ones chiefly: (a) a robust parser has to work in absence of information (hence it must learn from data); (b) a robust parser has to revise and to update the information.

In the last years, the idea that systematic and reliable acquisition on a large scale of linguistic information is the real challenge to NLP has been actually stressed. Moreover, currently available *corpora* make it is possible to build the core of a grammar and to increase the grammatical knowledge automatically from *corpora*. Two strategies vie with each other when it comes to approaching the specific problems of NLP we refer before: statistical versus rule-based strategies. From an engineering point of view, statistical extensions of linguistic theories have gained a vast popularity in the field of NLP: purely rule-based methods suffer from a lack of robustness in solving uncertainty due to overgeneration (if too many analyses are generated for a sentence) and undergeneration (if no analysis is generated for a sentence) [Bod98]. We think this 'lack of robustness' can be filled in the AI intention using abductive mechanisms that enlarge the deductive systems.

1.3 Abductive Mechanisms

We use the terms 'abductive mechanism' in a sense that may require a deeper explanation.

A deductive logical system typically offers a 'yes/no' answer to a closed question stated in the language of this logic. The two situations pointed out above can be found whenever we try to use a deduction system in realistic tasks of NLP:

(a) Lack of information in the *lexicon*. Thus, we have to use variables that do not belong to the logical language —$\alpha(X)$— for unknown values . An equivalent

problem in classical logic would be the following task: $p \vee q, p \rightarrow r, X \vdash r$. Stated in this way, it is not a deduction problem properly.
(b) A negative answer merely: $\nvdash \alpha$.

In both cases we could consider we have a theory (here, the *lexicon, L*) and a problem to solve: how the *lexicon* has to be modified and/or increased in order to obtain a deduction:
(a') $L \vdash Subs^{A}_{X}\alpha(X)$, where A belongs to the used logical language;
(b') $L \vdash \beta$, where β is obtained from α according to some constraints.

That is precisely what we have called 'abductive' problems (inasmuch as it is not a new rule, but new data that have to be searched for), and 'abductive mechanism' (as the method for its solution). One matter is the logical system on whose rules we justify a concrete yes/no answer to a closed question, and another matter is the procedure of searching for some answer, that admits to be labelled as abductive.

Our purpose is to introduce an abductive mechanism that enlarges LC in order to obtain a robust parser that can be fruitfully employed in realistic applications of NLP.[1]

1.4 State-of-the-Art in Categorial Grammar Learning

Large electronic *corpora* make the induction of linguistic knowledge a challenge. Most of the work in this field falls within the paradigm of classical automata and formal language theory [HU79], whether it uses symbolic methods, or statistical methods, or both.[2] As formal automata and language theory does not use the mechanisms of deductive logics, the used methods for learning a language from a set of data are not abductive or inductive mechanisms. Instead, they build an infinite sequence of grammars that converges in the limit.

This being the background, much of the work about learning Categorial Grammars deals with the problem of what classes of categorial grammars may be built from positive or negative examples in the limit.[3] This approach manages *corpora* that hold no tags at all, or that are tagged with the information of which item acts as functor and which item acts as argument.

The difference between those works and ours is that the former ones (a) have a wider goal—that of learning a whole class of categorial grammars from tagged *corpora*—, and (b) that they do not make use of any abductive mechanism, but follow the steps made in the field of formal language theory.

[1] Currently, LC seems to be relegated to an honourable *logical* place. It is far from constituting an indispensable methodology in NLP. Let us use the TMR Project *Learning Computational Grammars* as an illustration. This project "will apply several of the currently interesting techniques for machine learning of natural language to a common problem, that of learning noun-phrase syntax." Eight techniques are used. None is related to LC.
[2] Cfr. Gold [Gol67], Angluin [Ang80], [AS83], Bod [Bod98] and references therein.
[3] For this approach, cfr. Buszkowski [Bus87a], [Bus87b], Buszkowski and Penn [BP90], Marciniec [Mar94], Kanazawa [Kan98].

On the other hand, our work is (i) of a narrower scope—we are only interested in filling some gaps that the *lexicon* may have, or we want to change the category assigned by the *lexicon* to some lexical item when it does not lead to success —, and (ii) we use an abductive mechanism.

Finding the right category to assign to a lexical item is possible because we make use of a goal directed parsing algorithm that avoids infinite ramifications of the search tree trying only those categories that are consistent with the context.

2 A Parsing Algorithm Based on Lambek Calculus

2.1 Lambek Calculus

First, we introduce the Gentzen-style sequent formulation of LC. The underlying basic idea in the application of LC to natural language parsing is to assign a syntactic type (or category) to a lexical item. A concrete sequence of lexical items (words in some natural language) is grammatically acceptable if the sequent with these types as antecedent and the type s (sentence) as succedent is provable in LC.

The *language* of the (product-free) LC for CG is defined by a set of basic or atomic categories ($BASCAT$) -also called primitive types-, from which we form complex categories -also called types- with the set of right and left division operators $\{/, \backslash\}$:
If A and B are categories, then A/B, and $B\backslash A$ are categories.
We define a formula as being a category or a type.
In the following we shall use lower case latin letters for basic categories, upper case latin letters for whatever categories, lower case greek letters for non-empty sequences of categories, and upper case greek letters for, possible empty, sequences of categories.

The rules of LC are [Lam58]:

1. *Axioms:*

$$\frac{}{A \Rightarrow A}(Ax)$$

2. *Right Introduction: /R, \R*

$$\frac{\gamma, B \Rightarrow A}{\gamma \Rightarrow A/B}(/R) \qquad \frac{B, \gamma \Rightarrow A}{\gamma \Rightarrow B\backslash A}(\backslash R)$$

3. *Left Introduction: /L, \L*

$$\frac{\gamma \Rightarrow B \quad \Gamma, A, \Delta \Rightarrow C}{\Gamma, A/B, \gamma, \Delta \Rightarrow C}(/L) \qquad \frac{\gamma \Rightarrow B \quad \Gamma, A, \Delta \Rightarrow C}{\Gamma, \gamma, B\backslash A, \Delta \Rightarrow C}(\backslash L)$$

4. *Cut*

$$\frac{\gamma \Rightarrow A \quad \Gamma, A, \Delta \Rightarrow C}{\Gamma, \gamma, \Delta \Rightarrow C}(Cut)$$

It is required that each sequent has a non-empty antecedent and precisely one succedent category. The cut-rule is eliminable.

2.2 Automated Deduction in Lambek Calculus

Given a *lexicon* for a natural language, the problem of determining the grammatical correctness of a concrete sequence of lexical items (a parsing problem) becomes into a deductive problem in LC. Therefore, a parsing algorithm is just an LC theorem prover.
LC-theoremhood is decidable. However, LC typically allows many distinct proofs of a given sequent that assign the same meaning; this problem is called 'spurious ambiguity'. An efficient theorem prover has to search for (all) non-equivalent proofs only. There are in the literature two approaches to this problem, based on a normal form of proofs (Hepple [Hep90], König, Moortgat [Moo90], Hendriks [Hen93]) or on proof nets (Roorda [Roo91]). LC theorem prover we present is related to König's method [Kön89], but it solves problems which are proper to König's algorithm.

First, we introduce some definitions.

1. *Value and Argument Formulae*
   1.1. If $F = a$, then a is the value formula of F;
   1.2. If (i) $F = G/H$ or (ii) $F = H\backslash G$, then G is the value formula of F and H is the argument formula of F. In the case (i), H is the right argument formula; in the case (ii), H is the left argument formula.
2. *Value Path*
   The value path of a complex formula F is the ordered set of formulae $\langle A_1, \ldots, A_n \rangle$ such that A_1 is the value formula of F and A_j is the value formula of A_{j-1} for $2 \leq j \leq n$.
3. *Argument Path*
   The argument path of a complex formula F is the ordered set of formulae $\langle B_1, \ldots, B_n \rangle$ such that B_1 is the argument formula of F and B_j is the argument formula of A_{j-1}, for $2 \leq j \leq n$, and $\langle A_1, \ldots, A_n \rangle$ being the value path of F.
   The right (resp. left) argument path of a complex formula F is the ordered subset of its argument path owning right (resp. left) argument formulae only.
4. *Main Value Formula*
   A is the main value formula of a complex formula F whose value path is $\langle A_1, \ldots, A_n \rangle$ if and only if $A = A_n$.
   It follows that: (i) if A is a main value formula, then $A \in BASCAT$; (ii) every complex formula has exactly one main value formula.

2.3 The Algorithm

We now sketch the algorithm implemented in both C language and Prolog. We present the algorithm in a pseudo-Prolog fashion in order to provide an easier understanding. This is not Prolog, as we have simplified the management of data structures and other practical problems of the language. At the same time we assume a "try or fail" strategy of control like that of Prolog, as well as mechanisms of unification to build data structures. Self-evident procedures (`search_value`, etc.) are not included.

```
procedure proof
input:   data ⇒ target
output:  Proof tree if {⊢ data ⇒ target}, otherwise FAIL.
process:
    CASE data = target: RETURN {target⇒target (Ax)}
    CASE target = A/B: RETURN {proof(data,B⇒A) / data⇒A/B (/R)}
    CASE target = B\A: RETURN {proof(B,data⇒A) / data⇒B\A (\R)}
    CASE atomic(target):
        LET c := target
        LET [list_1, list_2, ..., list_n] := search_value(c in data)
        FOREACH list_i ∈ [list_1, list_2, ..., list_n] DO
            LET [α, F, β] := list_i
            LET [A_1, ..., A_k] := left_argument_path(c in F)
            LET [B_1, ..., B_m] := right_argument_path(c in F)
            LET tree_i := STACK reduce([ ], α, [A_k, ..., A_1])
                          WITH reduce([ ], β, [B_1, ..., B_m])
            IF tree_i = FAIL
               THEN CONTINUE
               ELSE RETURN {tree_i  c⇒c / data⇒c (|L)}
        END FOR
END procedure proof
```

```
procedure reduce
input:   ([acums], [data], [targets])
output:  proof tree if {⊩_LC acums, data ⇒ targets}, otherwise FAIL.
process:
    CASE acums = data = targets = [ ]: RETURN {— (empty)}
    CASE targets = [A]:
        RETURN proof(acums, data ⇒ A)
    OTHERWISE:
        CASE acums ≠ [ ] AND length(data) ≥ length(tail(targets)):
            LET tree := STACK proof(acums ⇒ head(targets))
                        WITH reduce(head(data),tail(data),tail(targets))
            IF tree ≠ FAIL
               THEN RETURN tree
               ELSE try next case
        CASE length(tail(data)) ≥ length(targets) - 1:
            RETURN reduce(acums+head(data),tail(data),tail(targets))
        OTHERWISE RETURN FAIL
END procedure reduce
```

2.4 Remarks on the Algorithm

(i) The `proof` procedure behaves as expected when input is an axiom.
(ii) The algorithm decomposes any target complex formula until it has to prove an atomic one, $c \in BASCAT$.

(iii) The `reduce` procedure is the main charasteristic of our algorithm. When we have to prove an atomic target, (i) we search for the formulae in the antecedent whose main value formula is the same as the atomic target $(F_1, \ldots, F_n)$; (ii) for each $F_i, 1 \leq i \leq n$, the left-hand side (resp. right-hand side) of the antecedent (with respect to F_i) and the left argument path of F_i (resp. right argument path) have to be cancelled out. The algorithm speeds up the deduction trying to satisfy the argument paths of F_i. The major advantages are obtained when the length of the sequence of data is long enough (note that a sentence in natural language may be up to 40 to 50 words long), and argument paths of the formulae are high. This property lies in the fact that the `reduce` procedure cares for still- not-consumed data and target formulae remaining to be proved. Efficient implementation for this algorithm has to avoid unnecessary calls to `proof` procedure from the `reduce` procedure, memorizing the proofs already tried.
(iv) `FAIL` may be regarded as an error propagating value. If any of the arguments of the proof-tree constructors —such as `STACK`, $(|L)$, $(/R)$, etc.— is `FAIL`, then resultant proof-tree is `FAIL`. A sensible implementation should be aware of this feature to stop the computational current step and to continue with the next one.

2.5 Properties of the Algorithm

(1) *The algorithm is correct*: If the output of `proof` procedure is not `FAIL`, then the proof tree constructed is a deduction of the input in LC.
Proof. Every rule we employ is a direct LC rule: axiom, $/R$, $\backslash R$. Note that the symbol $|L$ stands for successive applications of $/L$ and/or $\backslash L$. The conditions needed for applying each rule are exactly the same as they are required in LC. Hence, we can construct a proof tree in LC from the output of the `proof` procedure. □
(2) *The algorithm is complete*: If $\vdash_{LC}$ *data* $\Rightarrow$ *target*, then the output of the `proof` procedure is a proof tree.

The proof follows from (2.1) and (2.2) below:
(2.1) If there is no deduction in LC for $\gamma, B \Rightarrow A$, then there is no deduction in LC for $\gamma \Rightarrow A/B$. (Similarly for $B, \gamma \Rightarrow A$, and $\gamma \Rightarrow B\backslash A$)

Proof: Let us suppose that there is a proof tree, Π, in LC for $\gamma \Rightarrow A/B$.
Case 1: If every rule in Π is either a L-rule either an axiom, then we follow the deduction tree in a bottom-up fashion and we reach the sequent $A/B \Rightarrow A/B$. We can construct a proof Π' from Π in this way:

$$\frac{B \Rightarrow B \quad A \Rightarrow A}{A/B, B \Rightarrow A}(/L)$$

Next we apply the rules of Π over A/B that yield $\gamma \Rightarrow A/B$ in Π, and we obtain in Π': $\gamma, B \Rightarrow A$.
Case 2: If there is an application of /R in Π that yields

$$\frac{\delta, B \Rightarrow A}{\delta \Rightarrow A/B}(/R)$$

but it is not at the bottom of Π, we can postpone the application of the /R rule in Π' till remaining rules of Π have beeing applied, and so we have in Π' the sequent $\gamma, B \Rightarrow A$. □

We use these properties to decompose any complex succedent until we reach an atomic one.
(2.2) Let $c \in BASCAT$, and $\gamma = \gamma_1, \ldots, \gamma_n$ $(n > 0)$.
If $\vdash_{LC} \gamma \Rightarrow c$, then it exists some $\gamma_j, (1 \leq j \leq n)$ such that:
(i) c is the main value formula of γ_j;
(ii) $\Vdash_{LC} \gamma_1, \ldots, \gamma_{j-1} \Rightarrow \Phi$;
(iii) $\Vdash_{LC} \gamma_{j+1}, \ldots, \gamma_n \Rightarrow \Delta$;
(iv) A deduction tree for $\gamma \Rightarrow c$ can be reconstructed from (ii), (iii), and from the axiom $c \Rightarrow c$.
(Where $\langle A_1, \ldots, A_k \rangle$ is the left argument path of γ_j, $\Phi = \langle A_k, \ldots, A_1 \rangle$, and $\Delta = \langle B_1, \ldots, B_m \rangle$ is the right argument path of γ_j).

The symbol $\Vdash_{LC}$ stands for the fact that a sequence of formulae (data) proves a sequence of target formulae *keeping the order*. If we consider the Lambek Calculus with the product operator, $\bullet$, Φ and Δ can be constructed as the product of all A_i and all B_i respectively, and $\Vdash_{LC}$ can be substituted for $\vdash_{LC}$ in (ii), (iii).
Note that (ii) and (iii) state that $\gamma_1, \ldots, \gamma_{j-1}$ can be split up in k sequences of categories $(\alpha_k, \ldots, \alpha_1)$, and $\gamma_{j+1}, \ldots, \gamma_n$ can be split up in m sequences of categories $(\beta_1, \ldots, \beta_m)$ such that
(ii′) $\vdash_{LC} \alpha_n \Rightarrow A_n$, for $1 \leq n \leq k$;
(iii′) $\vdash_{LC} \beta_n \Rightarrow B_n$, for $1 \leq n \leq m$.
Proof:
Ad (i) No rule except an axiom allows to introduce c in the succedent. Following the deduction tree in a bottom-up fashion, successive applications of /L and \L are such that (a) the argument formulae in the conclusion turn into the succedent of the premise on the left; (b) the value formula remains as part of the antecedent of the premise on the right; (c) the succedent of the conclusion remains as the succedent of the premise on the right — note that this ordering of the premises is always possible. Therefore we will reach the sequent $c \Rightarrow c$ eventually, being c the main value formula of γ_j. □
This property allows us to restrict, without loss of completeness, the application of the L-rules to complex formulae whose main value formula is the same as the (atomic) target succedent.
Ad (ii) Let $\vdash_{LC} \gamma \Rightarrow c$. The only possibility of introducing A_n as a left argument formula of γ_j is from a L-rule. Hence, it exists some α_n such that $\vdash_{LC} \alpha_n \Rightarrow A_n$, because of $\alpha_n \Rightarrow A_n$ is the left-hand side premise of the L-rule. Otherwise, A_n together with c have to be introduced as an axiom, but the succedent is supposed to be an atomic type.
Note that we can first apply all L-rules for (/), followed by all L-rules for (\) —or vice versa—, whatever the formula may be. That follows from the theorems:
(a) $\vdash_{LC} (A\backslash(B/D))/C \Rightarrow ((A\backslash B)/D)/C$
(b) $\vdash_{LC} ((A\backslash B)/D)/C \Rightarrow (A\backslash(B/D))/C$

(c) $\vdash_{LC}\ C\backslash((D\backslash B)/A) \Rightarrow C\backslash(D\backslash(B/A))$
(d) $\vdash_{LC} C\backslash(D\backslash(B/A)) \Rightarrow\ C\backslash((D\backslash B)/A)$ □
Ad (iii) Similar to (ii). □
Ad (iv) Immediate from successive applications of /L and $\backslash L$. □
(3) *The algorithm stops.* For whatever sequence of data and target, the number of tasks is finite, and every step simplifies the complexity of the data and/or the target. □
(4) *The algorithm finds all different deduction and only once.*
If there are several formulae in γ such that (i)–(iii) hold, each case corresponds to a non equivalent deduction of $\gamma \Rightarrow c$.
The proof is based upon the fact that property (2.2) may be regarded as the construction of a proof-net for $\gamma \Rightarrow c$ (in the equivalent fragment of non-commutative linear logic). The axiom $c \Rightarrow c$ becomes the construction of an *axiom-link*, and the points (ii) and (iii) become the construction of the corresponding sub-proof-nets with no overlap. Different axiom-links produce different proof-nets. □

3 An Abductive Mechanism for NLP

We say a sequent is *open* if it has any unknown category instance in the antecedent and/or in the succedent; otherwise we say the sequent is *closed*. We use upper case latin letters from the end of the alphabet (X, Y, Z) for non-optional unknown categories, and X^*, Y^*, Z^* for optional unknown categories.

3.1 Learning and Discovery Processes

We would consider two abductive mechanisms that we shall call *learning* and *discovery* processes, depending on the form of the target sequent. Discovery processes are related to tasks involving open sequents; learning processes are related to tasks involving closed sequents.

1. Given a closed sequent, we may subdivide the possible tasks into:
   (a) Grammatical correctness: to check either or not a sequence of data yields a target, merely. This is the normal use of LC.
   (b) If a closed sequent is not provable, we can introduce a procedure for learning in two ways: according to data priority or according to target priority.
      i. If we have certainty about data, and a closed target is not provable from them, we remove the given target and we search for a (minimum) new target that may be provable from data. We need the target to be a minimum in order to avoid the infinite solutions produced by the type-raising rule.
      ii. If we have certainty about target, and the set of closed data does not prove it, we remove data, by means of re-typing the necessary lexical items, in such a way that the target becomes provable from these new data.

iii. If we have certainty about data and about target, we could consider the sequence as a linguistic phaenomenon that falls beyond a context free grammar, ellipsis, etc.

In both cases (b.i) and (b.ii) we can appropriately say that we learn new syntactic uses. Moreover, in case (b.ii) we carry out a revision of the *lexicon*.

2. An open sequent is related to discovery tasks. In a sense, every discovery task is also susceptible of being considered as a learning one (or vice versa). However, we would rather prefer to differentiate them by pointing out that they are based on formal features of the sequents.

3.2 The Abductive Mechanism

The objectives we pointed out above need the parsing algorithm —hereafter, $\mathcal{LC}$— to be enlarged using an abductive mechanism —hereafter, $\mathcal{ACG}$, Abductive Categorial Grammar— for handling open sequents and removing types if necessary. $\mathcal{ACG}$ manages:
(i) input sequences either from *corpora* or users;
(ii) information contained in the *lexicon*;
(iii) data transfer to $\mathcal{LC}$;
(iv) input adaptation and/or modification, if necessary;
(v) output of $\mathcal{LC}$;
(vi) request for a choice to the user;
(vii) addition of new types to the *lexicon* —its update.

What we have called an abductive mechanism has to do with the point (iv) most of all. We sketch only its main steps for taking into account the learning and discovery processes. Similarly to the parsing algorithm (2.3.), we present the procedure in a pseudo-prolog fashion.

procedure learning
input: $(data \Rightarrow target)(A)$
such that $\nvdash_{LC} data \Rightarrow target, \quad closed(data), \quad closed(target)$
output: substitution $\{A := B\}$
such that $\vdash_{LC} (data \Rightarrow target)\{A := B\}$
process:
CASE certainty_about_target:
LET $[A_1, \ldots, A_n] := data$
FOREACH $A_i \in [A_1, \ldots, A_n]$ **DO**
LET $new\_data := [\ldots, A_{i-1}, X_i, A_{i+1}, \ldots]$
$\{X_i := B_i\} :=$ **discovering** $new\_data \Rightarrow target$
END FOR
RETURN $\{A_1 := B_1, \ldots, A_n := B_n\}$
CASE certainty_about_data:
$\{X := B\} :=$ **discovering** $data \Rightarrow X$
RETURN $\{A := B\}$
END procedure learning

```
procedure discovering
input: (data ⇒ target)(X)
output: {X := B}
        such that  ⊢_LC (data ⇒ target){X := B}
process:
        CASE open_target: data ⇒ X
             IF data = [B]
             THEN RETURN {X := B}
             IF data = [F_1, ..., F_n]
             THEN FOREACH F_i(1 ≤ i ≤ n),  F_i ∉ BASCAT, DO
                  LET c_i := search_value(F_i)
                  {Y_i* := B_i, Z_i* := C_i} := new_proof Y_i*, data, Z_i* ⇒ c_i
             END FOR
             RETURN {X_1 := B_1\c_1/C_1, ..., X_n := B_n\c_n/C_n}
        CASE open_data:  data(X_1, ..., X_n) ⇒ target
             IF data = [X]
             THEN RETURN {X := target}
             FOREACH X_i(1 ≤ i ≤ n) DO
                  LET [F_1, ..., F_{i-1}, X_i, F_{i+1}, ..., F_n] := data
                  LET c := target
                  LET new_data := [F_1, ..., Y*\c/Z*, ..., F_n]
                  {X_i := B_i\c/C_i} := new_proof new_data ⇒ target
             END FOR
             RETURN {X_1 := B_1\c/C_1, ..., X_n := B_n\c/C_n}
END procedure discovering
```

3.3 Remarks on $\mathcal{ACG}$

The old `proof` procedure (2.3) has to be adapted to a `new_proof` one. To achieve this goal, we make two main changes: (a) the old `proof` procedure was built to work with closed sequents and now it should be able to deal with open ones; (b) the old `proof` procedure was initially designed to return a *proof tree* but it should now return the substitution that makes the open sequent provable.

The old `proof` algorithm may work with open sequents, behaving as an abductive mechanism, if we consider the (=) operator as *unification*. It is well known that the unification algorithm produces the substitution we are looking for.

Two major changes come (a) from the `search_value`(c `in` *data*) procedure, and (b) from the `reduce` procedure.

(a) The `search_value` procedure was considered to be self-evident, but now it needs further explanations inasmuch as unknown data or targets are present. What does it mean a value occurrence of X in Y? We will discuss the change in the process that considers a formula to be the main value of another one.

```
procedure search_value
input: (Formula from data, target formula)
```

```
output: ([right_argument_path],target formula,[left_argument_path]) or FAIL
   CASE closed data (F) and closed target (c):
      CASE F = c: RETURN ([ ], c, [ ])
      CASE F = B\A: RETURN ([B] + γ, c, δ)
                    where (γ, c, δ) := search_value(A, c)
      CASE F = A/B: RETURN (γ, c, [B] + δ)
                    where (γ, c, δ) := search_value(A, c)
      OTHERWISE RETURN FAIL
   CASE closed data (F) and open target (X):
      CASE F = c: RETURN X := c
      CASE F = B\A: RETURN STACK F
                    WITH search_value(A, c)
      CASE F = A/B: RETURN STACK F
                    WITH search_value(A, c)
   CASE open data (Y) and closed target (c): RETURN ([ ], Y := c, [ ])
   OTHERWISE RETURN FAIL
end procedure search_value
```

(b) Unknown categories may be either basic or complex ones. A treatment of the second case is rather difficult and it forces us to introduce constraints for bounding the search. We have to decide the upper bound of the complexity; i.e. X may be $A\backslash c/B$, or $A_1\backslash A_2\backslash c/B_1/B_2$, etc. The `reduce` procedure requires some adaptations for working with optional categories. Optional categories are matched only if they are needed in the proof.

```
CASE X* in target:
   IF data = [ ]
      THEN X* := [ ]
   ELSE X* := X
CASE X* in data
   IF target = [ ]
      THEN X* := [ ]
   ELSE
      LET [F1, ... , X*, ... , Fn] := data
      IF proof [F1, ... , Fn] ⇒ target ≠ FAIL
         THEN X* := [ ]
      ELSE X*:= new_proof [F1, ... , X, ... , Fn] ⇒ target
```

Finally, let us note that type-raising rules yield sequents like following: $A \Rightarrow X/(A\backslash X)$ or $A \Rightarrow (X/A)\backslash X$ —where A and X are whichever formulae— that are provable in LC. The basic (deductive) `proof` algorithm is complete and has no problem with the proof of such sequents, although some LC parsing algorithms in the literature (mainly natural deduction based ones) are not complete because of the type-raising rules are not provable in them. Regarding our `new_proof` algorithm, the problem arises when it works as an abductive process in which X, the target consequent, is unknown; then it may be regarded as atomic or as a

complex one. To regard it as atomic —our choice— causes no trouble but makes the type rising rule not provable (if the consequent is unknown). If we consider the possibility of an unknown consequent to be complex, then it may yield an endless loop. In fact, the type raising rule allows us to infer an endless number of more and more complex types.

3.4 Running $\mathcal{ACG}$

Example 1:
Data: "John loves".
Initial state of the *lexicon*:
John = np
loves = $np\backslash s/np$
Sketch of the abductive process:
(1) `proof` $(np, np\backslash s/np \Rightarrow s)$ = `FAIL`
(2) Certainty about data:
(2.1) $np, np\backslash s/np \Rightarrow X$
(2.2) $X := Y^*\backslash s/Z^*$
(2.3) $Y^*, np, np\backslash s/np, Z^* \Rightarrow s$
(2.4) $Y^*, np \Rightarrow np$; $Z^* \Rightarrow np$
(2.5) $Y^* := [\,]$; $Z^* := np$; $X := s/np$
Output:
• John loves = s/np
(3) Certainty about target:
(3.1) $X, np\backslash s/np \Rightarrow s$
(3.2) $X := s/Y^*$
(3.3) $np\backslash s/np \Rightarrow Y^*$
(3.4) $Y^* := np\backslash s/np$; $X := s/(np\backslash s/np)$
Output:
• John = $s/(np\backslash s/np)$
(3.5) $np, X \Rightarrow s$
(3.6) $X := Y^*\backslash s/Z^*$
(3.7) $np, Y^*\backslash s/Z^* \Rightarrow s$
(3.8) $np \Rightarrow Y^*$;
(3.9) $Y^* := np$; $Z^* := [\,]$
(3.10) $X := np\backslash s$
Output:
• loves = $np\backslash s$
(4) Certainty about data and target:
Output:
• John loves $\boxed{\text{X}}$ = $np, np\backslash s/np, \boxed{np} \Rightarrow s$.
Example 2:
Data: "someone bores everyone".
Initial state of the *lexicon*:
someone = ? (unknown)
bores = $np\backslash s/np$

everyone = ? (unknown)
$X, np\backslash s/np, Z \Rightarrow s$
Sketch of the abductive process:
(1)$X := s/Y^*$; $s/Y^*, np\backslash s/np, Z \Rightarrow s$
(1.1) $np\backslash s/np, Z \Rightarrow Y^*$
(1.2) $Y^* := Y_1^*\backslash s/Y_2^*$
(1.3) $Y_1^*, np\backslash s/np, Z, Y_2^* \Rightarrow s$
(1.4) $Y_1^* \Rightarrow np$
(1.5) $Z, Y_2^* \Rightarrow np$
(1.6) $Y_1^* := np$; $Z := np$; $Y_2^* := [\,]$
Output:
• someone = $s/(np\backslash s)$
• everyone = np
(2) $X \Rightarrow np$; $Z \Rightarrow np$
(2.1) $X := np$; $Z := np$
Output:
• someone = np
(3) $Z := Y^*\backslash s$; $X, np\backslash s/np, Y^*\backslash s \Rightarrow s$
(3.1) $X, np\backslash s/np \Rightarrow Y^*$
(3.2) $Y^* := Y_1^*\backslash s/Y_2^*$
(3.3) $Y_1^*, X, np\backslash s/np, Y_2^* \Rightarrow s$
(3.4) $Y_1^*, X \Rightarrow np$
(3.5) $Y_2^* \Rightarrow np$
(3.6) $Y_1^* := [\,]$; $X := np$; $Y_2^* := np$
Output:
• everyone = $(s/np)\backslash s$
State of the *lexicon* after runing $\mathcal{ACG}$:
someone = np, $s/(np\backslash s)$
bores = $np\backslash s/np$
everyone = np, $(s/np)\backslash s$

References

[Ang80] Angluin, D. Inductive Inference of Formal Languges from Positive Data. *Information and Control* **45** (1980), 117–135.

[AS83] Angluin, Dana and Smith, Carl H. Inductive Inference: Theory and Methods. *Computing Surveys* **15** (1983), 237–269.

[Bod98] Bod, R. *Beyond Grammar. An Experience-Based Theory of Language*. CSLI Publications, 1998.

[Bus87a] Buszkowski, W. Solvable Problems for Classical Categorial Grammars. *Bulletin of the Polish Academy of Sciences: Mathematics* **35**, (1987) 507–516.

[Bus87b] Buszkowski, W Discovery Procedures for Categorial Grammars. In E. Klein and J. van Benthem, eds. *Polimorphisms and Unifications*. University of Amsterdam, 1987.

[Bus97] Buszkowski, W. Mathematical linguistics and proof theory. In Benthem and Meulen [VBTM97], pages 683–736.

[BP90] Buszkowski, W. and Penn, G. Categorial Grammars Determined from Linguistics data by unification. *Studia Logica* **49**, (1990) 431–454.

[Gol67] Gold, E. M. Language Identification in the Limit. *Information and Control* **10**, (1967) 447–474.

[Hen93] Hendriks, H. *Studied Flexibility. Categories and Types in Sybtax and Semantics*. PhD Thesis, University of Amsterdam, 1993.

[Hep90] Hepple, M. *The Grammar and Processing of Order and Dependency: a Categorical Approach*. PhD. Thesis, University of Edinburgh, 1990

[HU79] Hopcroft, John E. and Ullman, Jeffrey D. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[Kan98] Kanazawa, M. *Learnable Classes of Categorial Grammars*. CSLI Publications, 1998.

[Kön89] König, E. Parsing as Natural Deduction. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics*. Vancouver, 1989.

[Lam58] Lambek, J. The Mathematics of Sentence Structure. *American Mathematical Monthly* **65** (1958), 154–169.

[Mar94] Marciniec, J. Learning Categorial Grammars by Unification with negative constraints. *Journal of Applications of Non-Classical Logics* **4** (1994), 181–200.

[Men95] Menzel, W. Robust Processing of Natural Language. cmp-lg/9507003.

[Moo90] Moortgat, M. Unambiguous Proof Representation for the Lambek Calculus. In *Proceedings of the Seventh Amsterdam Colloquium*, 1990.

[Moo97] Moortgat, M. Categorial type logics. In Benthem and Meulen [VBTM97], pages 93–178.

[Plot71] Plotkin, G.D. *Automatic Methods of Inductive Inference*. PhD Thesis, Edimburgh University, August 1971.

[Roo91] Roorda, D. *Resource Logics: Proof-theoretical Investigation*. PhD Thesis, University of Amsterdam, 1991.

[Ste92] Stede, M. The search for robustness in natural language understanding. *Artificial Intelligence Review*, **6** (1992) 383-414

[VBTM97] Van Benthem, J. and Ter Meulen, A., Editors. *Handbook of Logic and Language*. Elsevier Science B.V. and MIT Press, 1997.