Proof Automation for Type-Logical Grammars

Houda Anoun, Pierre Castéran, Richard Moot

*Labri, Université Bordeaux 1, and Inria Futurs (*Signes*)*

This document is written as a support for the course with the same title given at ESSLLI'2004 *by Richard Moot and Pierre Castéran*

CONTENTS

INTRODUCTION

Over the last few years, the field of type-logical grammars has matured to give us a flexible, expressive logical theory of natural language (Moortgat 1997, Morrill 1994). In this theory, parsing a phrase corresponds to finding a proof in this logical theory.

Because of the complexity of the mathematical objects to be manipulated – and the possibilities of human errors during this manipulation — the use of automated and semi-automated theorem provers seems to suggest itself.

In these notes, we will look at two such tools: first at Grail (Moot 1998) an automated theorem proved designed for designing and testing type-logical grammars, then at Coq (Bertot & Castéran 2004) a proof assistant based on higher-order logic, which we will use to prove meta-properties of type-logical grammars.

CHAPTER 1

TYPE-LOGICAL GRAMMARS

Type-logical grammars are a logic-based grammar formalism with several pleasant properties, such as effective learning algorithms (Buszkowski & Penn 1990) and a transparent λ term semantics thanks to the Curry-Howard isomorphism (van Benthem 1995).

In these notes, however, we will focus on two other properties: a nice proof theory in the form of graph, which is especially suited for automatically proving theorems in the system and the possibility to reason and prove properties *about* the system, which we will do in the next chapter.

In this chapter we will introduce type-logical grammars, and two logical calculi for it: natural deduction and proof nets. We will gradually introduce more detail and end with a linguistic example using all the constructs of the formalism.

1.1 AB Grammars

The simplest categorial grammars are those introduced by Ajdukiewicz and Bar-Hillel (Ajdukiewicz 1935, Bar-Hillel 1964), which we will call **AB** grammars in these notes. While note yet type-*logical* grammars, they will allow us to introduce some of the basic notions and vocabulary of the grammars we're interested in.

Our lexicon assigns formulas to words; atomic formulas, like *np* for noun phrase, *n* for noun and *s* for sentence and complex formulas of the form A/B and $B\backslash A$. A formula A/B (resp. $B\backslash A$) looks to the right (resp. left) for a B formula to produce an A formula.

For example, we can say that 'arthur' is a word of type *np*, which we will write as:

$$\text{arthur} \vdash np$$

To formalize this behavior, we introduce the following two rules of derivation.

$$\frac{X \vdash A/B \quad Y \vdash B}{X \circ Y \vdash A}\ [/E] \qquad \frac{Y \vdash B \quad X \vdash B\backslash A}{Y \circ X \vdash A}\ [\backslash E]$$

The $[/E]$ rule says that if X is an expression (lexical or complex) of type A/B and if Y is an expression of type B then the combination of these expressions $X \circ Y$ is of type A.

With just this simple calculus, we can derive sentences like the following.

$$\dfrac{\text{arthur} \vdash np \qquad \dfrac{\text{dismembers} \vdash (np\backslash s)/np \qquad \dfrac{\text{the} \vdash np/n \qquad \dfrac{\text{black} \vdash n/n \quad \text{knight} \vdash n}{\text{black} \circ \text{knight} \vdash n}\ [/E]}{\text{the} \circ (\text{black} \circ \text{knight}) \vdash np}\ [/E]}{\text{dismembers} \circ (\text{the} \circ (\text{black} \circ \text{knight})) \vdash np\backslash s}\ [/E]}{\text{arthur} \circ (\text{dismembers} \circ (\text{the} \circ (\text{black} \circ \text{knight}))) \vdash s}\ [\backslash E]$$

The Grail theorem prover uses an alternative form of representing proofs, which are called *proof nets*. An example lexicon for a type-logical proof net system is given in Figure 1.1. We have simple lexical entries, like 'Arthur' and 'rabbit', which simply assign a syntactic category to a word, but also more complex lexical entries, like 'the' which combines with a syntactic expression of category n to its right to form a syntactic expression of category np. Similarly, the transitive verb 'throws' combines with an np to its right and with an np to its left to form an expression of type s.

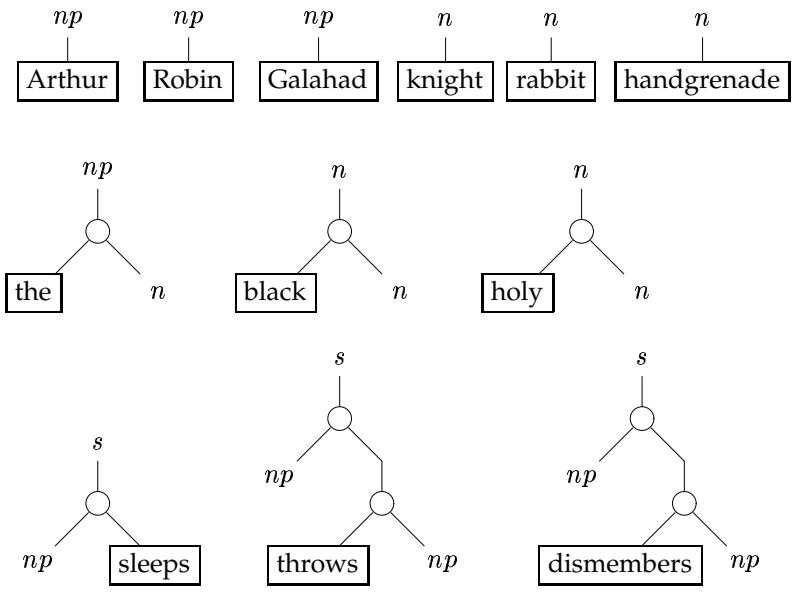

Figure 1.1: Some simple lexical graphs

Forming a proof net consists simply of connecting the atomic categories to produce a tree.

1.2 Categorial Grammars

Lambek (1958) was the first to turn the simple system we have seen so far into a

$$\frac{\begin{array}{c}[\mathrm{x} \vdash A]^n \quad [\mathrm{y} \vdash B]^n \\ \vdots \\ X \vdash A \bullet B \qquad Z[\mathrm{x} \circ \mathrm{y}] \vdash C\end{array}}{Z[X] \vdash C}\ [\bullet E]^n \qquad \frac{X \vdash A \quad Y \vdash B}{X \circ Y \vdash A \bullet B}\ [\bullet I]$$

$$\frac{X \vdash A/B \quad Y \vdash B}{X \circ Y \vdash A}\ [/E] \qquad \frac{\begin{array}{c}[\mathrm{x} \vdash B]^n \\ \vdots \\ X \circ \mathrm{x} \vdash A\end{array}}{X \vdash A/B}\ [/I]^n$$

$$\frac{Y \vdash B \quad X \vdash B\backslash A}{Y \circ X \vdash A}\ [\backslash E] \qquad \frac{\begin{array}{c}[\mathrm{x} \vdash B]^n \\ \vdots \\ \mathrm{x} \circ X \vdash A\end{array}}{X \vdash B\backslash A}\ [\backslash I]^n$$

Table 1.1: The natural deduction calculus **NL**

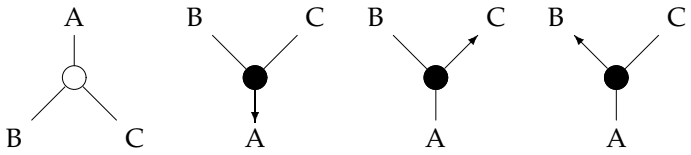

Figure 1.2: All binary graph constructors

logic. We will be principally interested in the non-associative Lambek calculus **NL** (Lambek 1961).

The rules of the calculus are given in Table 1.1. Note the presence of a new connective: an expression $A \bullet B$ is simply the composition of an A and B expression with A to the immediate left of B.

The rule $[/I]$ indicates that if we use a hypothesis x of type B to derive an expression where this x appears at the immediate right of some expression X, then we can conclude that this expression X must be of type A/B.

This move to logic is not just formally desirable, it also moves us closer to Montague-style semantics. Instead of assigning a generalized quantifier like someone a simple *np* type, we can now give it a higher order type $s/(np\backslash s)$, that is a type which gives a sentence whenever it finds a sentences missing an *np* to its right. Similarly, we assign a reflexive like 'himself' not an *np* type but $((np\backslash s)\ np)\backslash(np\backslash s)$, that is a word looking for a transitive verb to its left to produce an intransitive verb. This not only gets the semantics right, but also prevents sentences like '$*$himself sleeps' from being grammatical.

In the proof net calculus, moving to **NL** means allowing the constructors shown in Figure 1.2 for our lexical graphs.

In what follows, we will call the downward branching constructor, the *main* constructor; it has no constraints associated with its use.

The upward branching constructors, which we will call *auxiliary* and which we draw with a black center, denote *constraints* on the use of their lexical en-

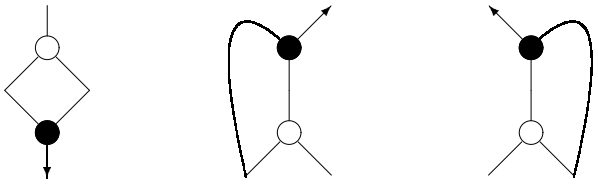

Figure 1.3: Graph contractions

try, in a sense to be made precise later. The only difference between the three auxiliary constructors is which of the three points it connects is the output, as indicated by the arrow.

Because the auxiliary constructors have more than one parent node, they prevent the graph we are constructing from being a *rooted* tree. To remove these auxiliary constructors, we define the graph contractions shown in Figure 1.3 on the graphs in our system, one for each of the auxiliary constructors. Whenever we find one of these three configurations of constructors, we can contract them to a single point.

Note that all contractions are of the same general form: they combine an auxiliary constructor with a main constructor on both ends not marked by the arrow and in a way which respects the up-down and left-right ordering of the nodes.

Also note that drawing these graphs on a plane sometimes requires us to bend one of the connections, because we want to keep all up-down and left-right distinctions explicit in the graph. These bends disappear if we draw the graphs on a cylinder.

A graph of this form is called a *proof net* iff it contracts to a tree consisting only of main constructors. Only proof nets correspond to derivations in **NL**.

We can use the auxiliary constructors to assign quantifiers like 'someone' the lexical graph given in Figure 1.4 on the left; this corresponds directly to the type we assigned to it before, but we refer the reader to (Moot & Puite 2002) to see about this correspondence in full formal detail. In the same Figure, a lexical graph for 'himself' is shown.

This lexical entry indicates that 'someone' selects an s to produce an s, where we can use an np inside this s, subject to the condition that the special constructor can be contracted according to Figure 1.3. The difference between the assignment of 'someone' to that of simple np's like 'Arthur' is that the assignment above allows 'someone' to take scope at sentence level as a generalized quantifier.

We can derive 'someone sleeps' as a well-formed expression of type s by the derivation shown in Figure 1.5. First, we connect the bottom s of 'someone' to the s of sleeps, which results in the structure on the left. After identifying the two np nonterminals, the structure will look as shown in the middle of Figure 1.5. Note that this structure is of the proper form to apply the contraction for the auxiliary constructor, as indicated by the dotted box around the redex. The resulting tree after the contraction is pictured on the right.

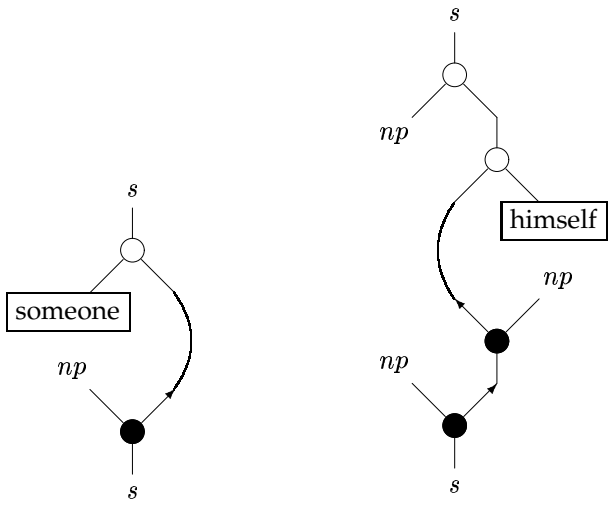

Figure 1.4: Lexical graph for a generalized quantifier and a reflexive

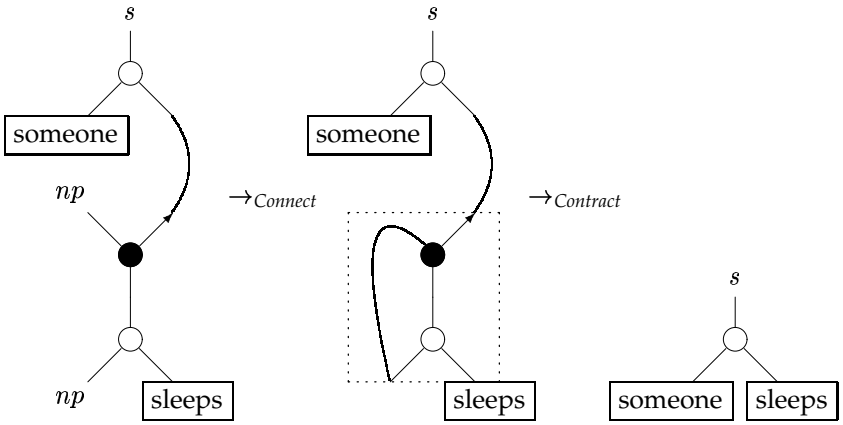

Figure 1.5: Derivation of 'someone sleeps'

Try to form a valid contraction of an example using the graph for 'himself' shown in Figure 1.4 yourself.

1.3 Unary Connectives

Unary connectives are introduce in type-logical grammars to play a role not unlike features in other grammar formalisms (Heylen 1999). The natural deduction rules for these connectives are shown in Figure 1.2.

In the proof net calculus, it corresponds to allowing our lexical graphs to have unary branches, which look as shown in Figure 1.6.

As with the binary constructors, the only difference between the unary auxiliary constructors is in the arrow which indicates the output connection. We

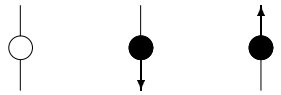

Figure 1.6: Unary constructors

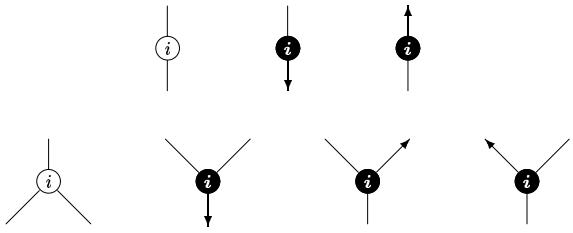

Figure 1.7: Constructors with mode information

can contract a main and an auxiliary unary connector if they are connected at the point which is not the output of the auxiliary constructor, just like with the binary constructors.

Secondly, we allow our constructors to have different modes of composition by writing an index i, out of a finite set of possible indices I, inside the constructor, as shown in Figure 1.7.

Example modes are, for example *nom* for nominative and *acc* for accusative. Using this idea, we can refine our lexical type assignments for transitive verbs to the following.

$$(\Diamond_{nom}\Box^{\downarrow}_{nom}np\backslash s)/\Diamond_{acc}\Box^{\downarrow}_{acc}np$$

That is, we utilize the relations between the unary operators which allow us to derive $A \vdash \Diamond_i\Box^{\downarrow}_i A$ for all formulas A and modes i (try to prove this using the natural deduction rules). This means we can still use our old *np* types for words like 'Arthur', which are indifferent to the case assigned by the verb. But assigning an accusative pronoun like 'him' the type $\Diamond_{acc}\Box^{\downarrow}_{acc}np$ will now prevent a sentence like '$*$ him sleeps' from being derivable.

Try to figure out which lexical graphs for verbs and pronouns will produce this behavior for proof nets.

1.4 Structural Rules

The logic we have just described is still quite limited in the type of reasoning it allows, this because of the total absence of structural rules (even associativity!) from the logic.

In order to introduce some of the flexibility necessary to capture linguistic phenomena like, for example, long-distance dependencies and medial extraction, we will now add *controlled* versions of structural rules to our logic.

Hypothesis

$$\mathrm{x} \vdash A$$

Binary Connectives

$$\frac{\begin{array}{c}[\mathrm{x} \vdash A]^n \quad [\mathrm{y} \vdash B]^n \\ \vdots \\ X \vdash A \bullet_i B \qquad Z[\mathrm{x} \circ_i \mathrm{y}] \vdash C\end{array}}{Z[X] \vdash C}\ [\bullet E]^n \qquad \frac{X \vdash A \quad Y \vdash B}{X \circ_i Y \vdash A \bullet_i B}\ [\bullet I]$$

$$\frac{X \vdash A/_i B \quad Y \vdash B}{X \circ_i Y \vdash A}\ [/E] \qquad \frac{\begin{array}{c}[\mathrm{x} \vdash B]^n \\ \vdots \\ X \circ_i \mathrm{x} \vdash A\end{array}}{X \vdash A/_i B}\ [/I]^n$$

$$\frac{Y \vdash B \quad X \vdash B\backslash_i A}{Y \circ_i X \vdash A}\ [\backslash E] \qquad \frac{\begin{array}{c}[\mathrm{x} \vdash B]^n \\ \vdots \\ \mathrm{x} \circ_i X \vdash A\end{array}}{X \vdash B\backslash_i A}\ [\backslash I]^n$$

Unary Connectives

$$\frac{\begin{array}{c}[\mathrm{x} \vdash A]^n \\ \vdots \\ X \vdash \Diamond_i A \quad Z[\langle \mathrm{x} \rangle^i] \vdash C\end{array}}{Z[X] \vdash C}\ [\Diamond E]^n \qquad \frac{X \vdash A}{\langle X \rangle^i \vdash \Diamond_i A}\ [\Diamond I]$$

$$\frac{X \vdash \Box_i^{\downarrow} A}{\langle X \rangle^i \vdash X}\ [\Box^{\downarrow} E] \qquad \frac{\langle X \rangle^i \vdash A}{X \vdash \Box_i^{\downarrow} A}\ [\Box^{\downarrow} I]$$

Structural Rules

$$\frac{\Gamma \ \vdash \ Z[\Xi'[X_1, \ldots, X_n]] \vdash C}{\Gamma \ \vdash \ Z[\Xi[X_{\pi_1}, \ldots, X_{\pi_n}]] \vdash C}\ [SR]$$

Table 1.2: The natural deduction calculus $\mathbf{NL}\Diamond_{\mathcal{R}}$

This allows us to key our structural rules to the presence or absence of certain modes, thereby preventing a collapse to full commutativity of the logic.

The full logic is given in Table 1.2, where the structural rules at the bottom allow us to replace a structure tree Ξ' by a structure tree Ξ provided they have the same leaves (though perhaps in a different order, as indicated by the permutation function π).

For the proof nets, the structural rules correspond to *structural conversions* which convert one tree of main constructors into another tree of main constructors with the same leaves. An example of a structural conversion, where 0 and

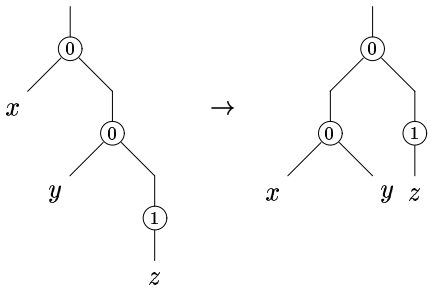

Figure 1.8: Example structural rule

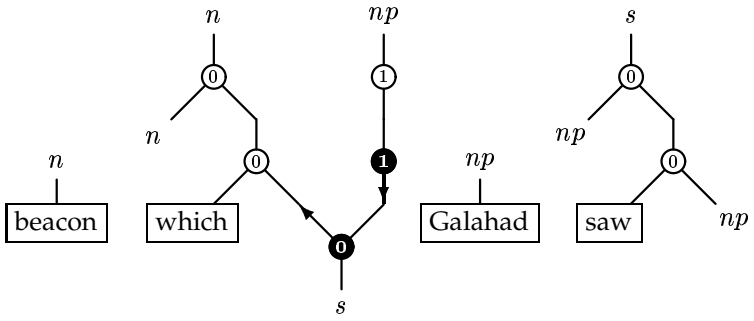

Figure 1.9: Lexical graphs for 'beacon which Galahad saw'

1 are elements of I, is shown in Figure 1.8.

1.5 Example: Putting it all together

We will now give an example using all the additions to the logic we have seen so far: unary connectives, mode information and structural rewrites.

Suppose we have the lexical trees shown in Figure 1.9 and want to derive 'beacon which Galahad saw' as an expression of type n. Note the tree to 'which', with the pair of unary modes attached to the np. We could contract this particular configuration immediately, but since the unary mode gives us access to the structural rewrite of Figure 1.8, we choose to connect the atomic formulas first.

After these connections, we end up with the graph shown in Figure 1.10. A different way to portray this same graph is shown in Figure 1.11. Note that we have just moved the two unary constructors down, in order to make the subgraph to which we are going to apply the structural conversion more clear and that, as noted before, if the graphs were to be portrayed on a cylinder, there would be no visible difference between the two graphs.

Note that we are still in a position to contract the unary connective, but that should we perform this contraction, it will be impossible to continue: no structural rewrites apply, and we will not be in the right configuration for the final contraction. However, we have the right to perform a structural rewrite

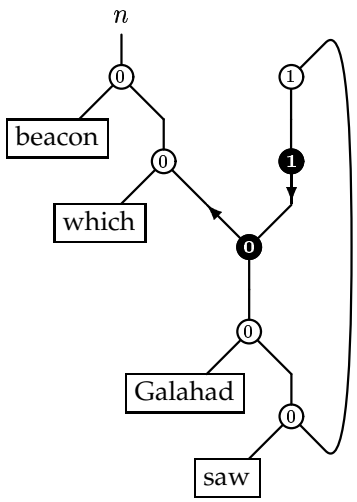

Figure 1.10: Graph after connecting all atomic formulas

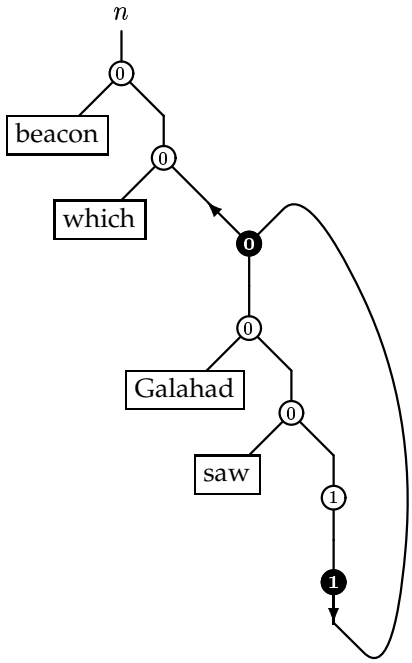

Figure 1.11: Graph after connecting all atomic formulas

here, which produces the graph shown in Figure 1.12.

Only now will we apply the unary contraction, producing the graph shown in Figure 1.13.

You will notice that we are now immediately in the right configuration to perform the final contraction which produces the tree shown in Figure 1.14.

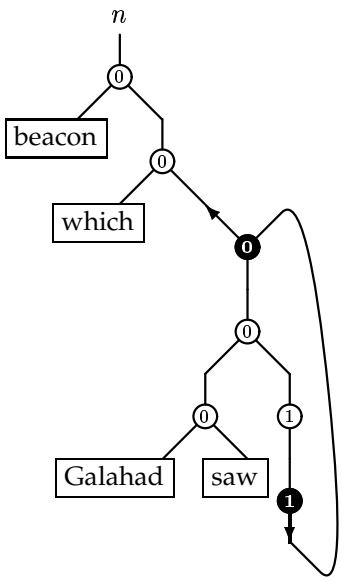

Figure 1.12: Graph after the structural rewrite

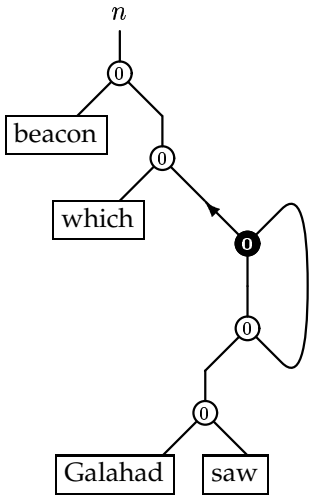

Figure 1.13: Graph after the unary contraction

1.6 Conclusion

We have very briefly presented the architecture of type-logical grammars and proof nets, like they are used in the Grail theorem prover. As with any formal system, the best way to understand them is to try out a few examples. We therefore recommend you use Grail to try out some of the provided example files and see if you can construct proof nets for the sentences there. Good luck!

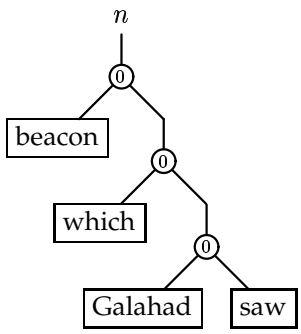

Figure 1.14: Tree after final contraction

CHAPTER 2

REASONING ABOUT CATEGORIAL TYPE GRAMMARS

Like many formalizations of natural languages, categorial grammars — and in particular multimodal grammars — depend on a lot of parameters : atomic types, modes, structural rules and lexicons. That complexity makes the study of categorial grammars very challenging.

It is thus desirable to express some properties regarding these parameters, in order to understand their influence on syntax and semantics, and to simplify the design of new grammars.

Since the formalism in Table 1.2 contains 12 rules (the last one parameterized by an arbitrary set of structural rules), proving these properties may be quite time-consuming. The *Icharate* toolkit, developed by Houda Anoun and Pierre Castéran, is a library of theories written in the *Coq* proof assistant (Bertot & Castéran 2004, The Coq development team 2004), which allows to prove and use such " meta-rules ".

These notes present how *Icharate* was designed, its current state after one year of development, its future evolution. We shall also discuss how mechanized deduction can play a role in computational linguistics besides fully automatic methods.

2.0.1 Rules for natural deduction

For our purpose, we shall introduce a slight change in notation for natural deduction. First, contexts are made from type formulas and not words[1] As in (Moortgat 1997), the operator for the composition of two contexts Γ_1 and Γ_2 is denoted by $(\Gamma_1\ ,\ \Gamma_2)_i$ instead of $\Gamma_1\ \circ_i\ \Gamma_2$. Figure 2.1 shows a partial view of the notations we use. Notice that the third column shows a linear presentation of the tree-like derivations.

[1] In the real *Coq* implementation, contexts contain both words and formulas, but we'll keep things simple for now.

name	rule	term notation
Axiom	$\dfrac{}{A \vdash_{\boldsymbol{R}} A}\ \mathbf{Ax}$	$\mathbf{Ax}(A)$
$\bullet$- introduction	$\dfrac{\dfrac{d_1}{\Gamma_1 \vdash_{\boldsymbol{R}} A_1} \quad \dfrac{d_2}{\Gamma_2 \vdash_{\boldsymbol{R}} A_2}}{(\Gamma_1\,,\,\Gamma_2)_i \vdash_{\boldsymbol{R}} A_1 \bullet_i A_2}\ \bullet_{\mathbf{I}}$	$\bullet_{\mathbf{I}}(i, \Gamma_1, A_1, \Gamma_2, A_2, d_1, d_2)$
/-introduction rule	$\dfrac{\dfrac{d}{(\Gamma\,,\,B)_i \vdash_{\boldsymbol{R}} A}}{\Gamma \vdash_{\boldsymbol{R}} A/_i B}\ /_{\mathbf{I}}$	$/_{\mathbf{I}}(i, \Gamma, A, B, d)$
\-introduction rule	$\dfrac{\dfrac{d}{(B\,,\,\Gamma)_i \vdash_{\boldsymbol{R}} A}}{\Gamma \vdash_{\boldsymbol{R}} B\backslash_i A}\ \backslash_{\mathbf{I}}$	$\backslash_{\mathbf{I}}(i, \Gamma, A, B, d)$
$\bullet$-elimination rule	$\dfrac{\dfrac{d_1}{\Delta \vdash_{\boldsymbol{R}} A \bullet_i B} \quad \dfrac{d_2}{\Gamma[(A\,,\,B)_i] \vdash C}}{\Gamma[\Delta] \vdash_{\boldsymbol{R}} C}\ \bullet_{\mathbf{E}}$	$\bullet_{\mathbf{E}}(i, \Gamma[\,], \Delta, A, B, C, d_1, d_2)$
...	...	...

Figure 2.1: Rules for natural deduction (as trees)

2.1 Simple derived rules

Let us begin our presentation with simple derived rules, obtained by a finite number of applications of the rules in Figure 2.1. The name of the following derived rules come from Michael Moorgat's contribution to the Handbook of Logic and Language (Moortgat 1997).

2.1.1 Rules without premises

The simplest examples are rules of the form $\Gamma \vdash_{\boldsymbol{R}} F$, where Γ and F contain first-order variables A, B, C ..., ranging over formulas. Such a rule must be considered as if preceded by universal quantification on A, B, C and R. If no mode decorates the operators /, \, etc., we consider that the rules are valid for any binary mode i and unary mode j, and all operators are decorated with either i or j.

Figures 2.2, 2.3 and 2.4 show derivations of three such rules. The reader will notice that getting these generic rules is not so different from performing an analysis like in Chapter 1.

$$\dfrac{\dfrac{\dfrac{}{A \vdash A}\ Ax \quad \dfrac{}{B \vdash B}\ Ax}{A,B \vdash A \bullet B}\ \bullet_{\mathbf{I}}}{A \vdash ((A \bullet B)/B)}\ /_{\mathbf{I}}$$

Figure 2.2: Proof of the co-application rule

2.1.2 Derived rules with premises

It is easy to derive rules of the following form :

$$\dfrac{\dfrac{}{B\bullet B\backslash A \vdash B\bullet B\backslash A}\,Ax \qquad \dfrac{\dfrac{}{B \vdash B}\,Ax \quad \dfrac{}{B\backslash A \vdash B\backslash A}\,Ax}{B, B\backslash A \vdash A}\,\backslash_{\mathbf{E}}}{B\bullet B\backslash A \vdash A}\,\bullet_{\mathbf{E}}$$

Figure 2.3: Proof of the application rule

$$\dfrac{\dfrac{}{A \vdash A}\,Ax \quad \dfrac{}{A\backslash B \vdash A\backslash B}\,Ax}{\dfrac{A, A\backslash B \vdash B}{A \vdash B/(A\backslash B)}\,/_{\mathbf{I}}}\,\backslash_{\mathbf{E}}$$

Figure 2.4: Proof of the lifting rule

$$\dfrac{\Gamma_1 \vdash A_1 \quad \dots \quad \Gamma_n \vdash A_n}{\Gamma \vdash A}$$

To derive such rules, one considers arbitrary derivations d_1 of $\Gamma_1 \vdash A_1$, … , d_n of $\Gamma_n \vdash A_n$, which we can use as subderivations to build a derivation of $\Gamma \vdash A$. A simple example is the "monotonicity of $\bullet$", shown in Figure 2.5.

$$\dfrac{\dfrac{\dfrac{d_1}{A \vdash B} \quad \dfrac{d_2}{C \vdash D}\,\bullet_{\mathbf{I}}}{A, C \vdash B\bullet D}\,\bullet_{\mathbf{E}}}{A\bullet C \vdash B\bullet D}$$

Figure 2.5: Proof of $\bullet$-monotonicity

2.1.3 Using structural postulates

Among conditional derived rules, we can consider schemes of the form: "if R contains the structural rule r, then $\Gamma \vdash_{\boldsymbol{R}} A$". An example (also quoted from Moortgat) is the following rule, called "Geach (secondary functor)":

$$\dfrac{\mathbf{L2}(a) \in R}{B/_a C \vdash_{\boldsymbol{R}} (A/_a B)\backslash_a (A/_a C)}$$

Its derivation is shown in Figure 2.6.

2.1.4 Multimodal derivations

2.1.4.1 Interaction postulates and meta-theorems

In the multimodal logic, we are able to define structural rules that allow communication between different modes of composition, those rules are called interaction principles. The principle **MA** (Figure 2.7) is an example of that kind of rules.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A/_aB \vdash_{\boldsymbol{R}} A/_aB}\ Ax \quad \dfrac{\dfrac{}{B/_aC \vdash_{\boldsymbol{R}} B/_aC}\ Ax \quad \dfrac{}{C \vdash_{\boldsymbol{R}} C}\ Ax}{(B/_aC\ ,\ C)_a \vdash_{\boldsymbol{R}} B}\ /_{\mathbf{E}}}{(A/_aB\ ,\ (B/_aC\ ,\ C)_a)_a \vdash_{\boldsymbol{R}} A}\ /_{\mathbf{E}}}{((A/_aB\ ,\ B/_aC)_a\ ,\ C)_a \vdash_{\boldsymbol{R}} A}\ L2\ a}{(A/_aB\ ,\ B/_aC)_a \vdash_{\boldsymbol{R}} A/_aC}\ /_{\mathbf{I}}}{B/_aC \vdash_{\boldsymbol{R}} (A/_aB)\backslash_a(A/_aC)}\ \backslash_{\mathbf{I}}$$

Figure 2.6: Proof of Geach rule

$$\dfrac{\Gamma[(\Delta_1\ ,\ (\Delta_2\ ,\ \Delta_3)_i)_j] \vdash C}{\Gamma[((\Delta_1\ ,\ \Delta_2)_j\ ,\ \Delta_3)_i] \vdash C}\ MA(i,j)$$

Figure 2.7: Interaction rule MA(i,j)

We can generalize the Geach rule seen in the previous section as follows :

$$\dfrac{\mathbf{MA}(i,j) \in R}{B/_iC \vdash_{\boldsymbol{R}} (A/_jB)\backslash_j(A/_iC)}$$

Its derivation is shown in Figure 2.8. Notice that **L2** is a particular case of

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A/_jB \vdash_{\boldsymbol{R}} A/_jB}\ Ax \quad \dfrac{\dfrac{}{B/_iC \vdash_{\boldsymbol{R}} B/_iC}\ Ax \quad \dfrac{}{C \vdash_{\boldsymbol{R}} C}\ Ax}{(B/_iC\ ,\ C)_i \vdash_{\boldsymbol{R}} B}\ /_{\mathbf{E}}}{(A/_jB\ ,\ (B/_iC\ ,\ C)_i)_j \vdash_{\boldsymbol{R}} A}\ /_{\mathbf{E}}}{((A/_jB\ ,\ B/_iC)_j\ ,\ C)_i \vdash_{\boldsymbol{R}} A}\ MA(i\ j)}{(A/_jB\ ,\ B/_iC)_j \vdash_{\boldsymbol{R}} A/_iC}\ /_{\mathbf{I}}}{B/_iC \vdash_{\boldsymbol{R}} (A/_jB)\backslash_j(A/_iC)}\ \backslash_{\mathbf{I}}$$

Figure 2.8: New Proof of Geach rule

MA, where i=j.

2.1.4.2 Using the unary operators

In the following we will show an example of the analysis of non-peripheral extraction using multimodal logic. Such a phenomenon is illustrated for instance in the expression '*whom Mary loves madly*' where the extracted element (object of the verb loves) is neither the leftmost nor the rightmost subconstituent within the relative pronoun's complement. Let us consider the structural rules $\mathbf{L}_\Diamond$ (Figure 2.9) and $\mathbf{P}_\Diamond$ (Figure 2.10), which allow restricted associativity and commutativity.

Using these structural principles, we are able to derive the following rule:

$$\dfrac{\mathbf{L}_\Diamond(i,j) \in R \quad \mathbf{P}_\Diamond(i,j) \in R}{(A\ ,\ ((A\backslash_iB)/_iC\ ,\ (A\backslash_iB)\backslash_i(A\backslash_iB))_i)_i \vdash_{\boldsymbol{R}} B/_i\Diamond_j\Box_jC}\ \text{extraction}$$

$$\frac{\Gamma[(\Delta_1\ ,\ (\Delta_2\ ,\ <\Delta_3>_j)_i)_i]\vdash C}{\Gamma[((\Delta_1\ ,\ \Delta_2)_i\ ,\ <\Delta_3>_j)_i]\vdash C}\ \mathbf{L}_\Diamond(i,j)$$

Figure 2.9: Interaction rule $\mathbf{L}_\Diamond$(i,j)

$$\frac{\Gamma[((\Delta_1\ ,\ \langle\Delta_3\rangle^j)_i\ ,\ \Delta_2)_i]\vdash C}{\Gamma[((\Delta_1\ ,\ \Delta_2)_i\ ,\ \langle\Delta_3\rangle^j)_i]\vdash C}\ \mathbf{P}_\Diamond(i,j)$$

Figure 2.10: Interaction rule $\mathbf{P}_\Diamond$(i,j)

This rule's derivation is shown in Figure 2.11

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A\vdash A}\ Ax \qquad \dfrac{\dfrac{\dfrac{}{\Diamond_j\Box_j C\vdash_{\boldsymbol{R}}\Diamond_j\Box_j C}\ Ax \qquad \dfrac{\dfrac{}{(A\backslash_i B)/_i C\vdash_{\boldsymbol{R}}(A\backslash_i B)/_i C}\ Ax \qquad \dfrac{\dfrac{}{\Box_j C\vdash_{\boldsymbol{R}}\Box_j C}\ Ax}{\langle\Box_j C\rangle^j\vdash_{\boldsymbol{R}} C}\ \Box_{\mathbf{E}}}{((A\backslash_i B)/_i C\ ,\ \langle\Box_j C\rangle^j)_i\vdash_{\boldsymbol{R}} A\backslash_i B}\ /_{\mathbf{E}}}{((A\backslash_i B)/_i C\ ,\ \Diamond_j\Box_j C)_i\vdash_{\boldsymbol{R}} A\backslash_i B}\ \Diamond_{\mathbf{E}}}{(((A\backslash_i B)/_i C\ ,\ \Diamond_j\Box_j C)_i\ ,\ (A\backslash_i B)\backslash_i(A\backslash_i B))_i\vdash_{\boldsymbol{R}}(A\backslash_i B)}\ \backslash_{\mathbf{E}}}{(A\ ,\ (((A\backslash_i B)/_i C\ ,\ \Diamond_j\Box_j C)_i\ ,\ (A\backslash_i B)\backslash_i(A\backslash_i B))_i)_i\vdash_{\boldsymbol{R}} B}\ \backslash_{\mathbf{E}}}{(A\ ,\ (((A\backslash_i B)/_i C\ ,\ (A\backslash_i B)\backslash_i(A\backslash_i B))_i\ ,\ \Diamond_j\Box_j C)_i)_i\vdash_{\boldsymbol{R}} B}\ \mathbf{P}_\Diamond(i\ j)}{((A\ ,\ ((A\backslash_i B)/_i C\ ,\ (A\backslash_i B)\backslash_i(A\backslash_i B))_i)_i\ ,\ \Diamond_j\Box_j C)_i\vdash_{\boldsymbol{R}} B}\ \mathbf{L}_\Diamond(i\ j)}{(A\ ,\ ((A\backslash_i B)/_i C\ ,\ (A\backslash_i B)\backslash_i(A\backslash_i B))_i)_i\vdash_{\boldsymbol{R}} B/_i\Diamond_j\Box_j C}\ /_{\mathbf{I}}$$

Figure 2.11: Non peripheral extraction rule

Using this rule and the lexicon of Figure 2.12, we can easily deduce a derivation of '*whom Mary likes madly*' $\vdash \mathrm{n}\backslash_a\mathrm{n}$. This is possible because the extracted element is decorated with unaries ($\Diamond_c\ \Box_c$ np) which enable a limited access to both associativity and commutativity (see Figure 2.13.)

Remark: Using such restricted structural rules does not allow us to derive ungrammatical expressions where the extracted element is in the subject position, as it is the case in '*(*whom likes Mary madly)*'.

2.1.4.3 Interaction principles with contraction

We will show here how can a restricted contraction rule be useful to analyze complex phenomena in natural languages. As an example of these rules, we consider the rule **MC** shown in Figure 2.14.

Using this rule, we are able to derive the backward crossed substitution rule (used as a combinator by Mark Steedman (Steedman 1996)):

$$\frac{\mathbf{MC}(i,j)\in R}{(A/_jB\ ,\ (A\backslash_i C)/_jB)_i\vdash_{\boldsymbol{R}} C/_jB}$$

The proof of this derived rule is given in Figure 2.1.4.3.

whom	Mary	likes	madly
$(n\backslash_a n)/_a(s/_a\Diamond_c \Box_c np)$	np	$(np \backslash_a s)/_a np$	$(np \backslash_a s)\backslash_a(np \backslash_a s)$

Figure 2.12: A lexicon

$$\dfrac{\dfrac{}{whom \vdash_{\boldsymbol{R}} (n\backslash_a n)/_a(s/_a\Diamond_c\Box_c np)}\ Ax \quad \dfrac{}{(Mary\,,\,(loves\,,\,madly)_a)_a \vdash_{\boldsymbol{R}} s/_a\Diamond_c\Box_c np}}{(whom\,,\,(Mary\,,\,(loves\,,\,madly)_a)_a)_a \vdash_{\boldsymbol{R}} n\backslash_a n}\ \begin{matrix} extraction \\ /_{\mathbf{E}} \end{matrix}$$

Figure 2.13: Application of the extraction theorem

Such a derived rule assures the analysis of parasitic gaps (Steedman 1996) which can be illustrated by the following relative clause '*which John will file _ without reading _*'. Given the lexicon of the Figure 2.16, we can easily show that the expression '*file without reading*' has the type of a transitive verb and thus deduce that the relative phrase above is a noun modifier.

This derivation, shown in Figure 2.17, uses the structural rule **(MC a c)** and also restricted associativity **(MA c a)**.

2.2 Using auxiliary computations

The derived rules we have seen until now were quite easy to build, by composing a finite number of rules (basic or already derived rules). However, some linguistic phenomena may require a variable amount of rule applications. In these cases, the associated derived rule will contain a call to an auxiliary recursive function whose work is to call these rules as many times as needed.

To illustrate this point, let us consider the phenomenon of *u*nbound dependencies (Hepple 1990, Carpenter 1998), i.e. constructions that bind two arbitrarily distant expressions within a sentence.

For instance, consider the three sentences below, where the symbol '_' represents the position of the extracted expressions:

(i) '*The boy* **whom** *Mary loves _ is nice*'

(ii) '*The boy* **whom** *John believes Mary loves _ is nice*'

(iii) '*The boy* **whom** *John believes Mary thinks the girl loves _ is nice*'

Let us consider the third sentence, with the lexicon shown in Figure 2.18. We want to prove that '*whom John believes Mary thinks the girl loves _ is nice*' has type $n\backslash_a n$. This problem can be reduced to showing that the context *((John ,*

$$\dfrac{\Gamma[((\Delta_1\,,\,\Delta_3)_j\,,\,(\Delta_2\,,\,\Delta_3)_j)_i]\vdash C}{\Gamma[((\Delta_1\,,\,\Delta_2)_i\,,\,\Delta_3)_j]\vdash C}\ MC(i,j)$$

Figure 2.14: Interaction rule MC(i,j)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{A/_jB \vdash_{\boldsymbol{R}} A/_jB}\,Ax \quad \dfrac{}{B \vdash B}\,Ax}{(A/_jB\,,\,B)_j \vdash_{\boldsymbol{R}} A}\,/_{\mathbf{E}} \quad \dfrac{\dfrac{}{(A\backslash_iC)/_jB \vdash_{\boldsymbol{R}} (A\backslash_iC)/_jB}\,Ax \quad \dfrac{}{B \vdash_{\boldsymbol{R}} B}\,Ax}{((A\backslash_iC)/_jB\,,\,B)_j \vdash_{\boldsymbol{R}} A\backslash_iC}\,/_{\mathbf{E}}}{((A/_jB\,,\,B)_j\,,\,((A\backslash_iC)/_jB\,,\,B)_j)_i \vdash_{\boldsymbol{R}} C}\,\backslash_{\mathbf{E}}}{((A/_jB\,,\,(A\backslash_iC)/_jB)_i\,,\,B)_j \vdash_{\boldsymbol{R}} C}\,MC(i,j)}{(A/_jB\,,\,(A\backslash_iC)/_jB)_i \vdash_{\boldsymbol{R}} C/_jB}\,/_{\mathbf{I}}$$

Figure 2.15: Derivation of S combinator

which	John	will	file	without	reading
$(n\backslash_a n)/_a(s/_c np)$	np	$(np\backslash_a s)/_a(np\backslash_a s)$	$(np\backslash_a s)/_c np$	$((np\backslash_a s)\backslash_a(np\backslash_a s))/_c np/_a pp$	pp

Figure 2.16: Lexicon for parasitic gaps

(believes , (Mary , (thinks ,((the, girl), loves))))), $<h : \Box_c\ np>_c$*)* can be derived in '**s**'.

We need to transform the structure of the context associated with the relative '*whom John believes Mary thinks the girl loves _ is nice*' in order to place the variable of type `np` representing '_' as the right hand side argument of `loves`. This can be done with several applications of the structural rule $\mathbf{L}_\Diamond(\mathtt{a},\mathtt{c})$ (Figure 2.9.) Figure 2.19 shows the first and last steps of this transformation.

Since the number of applications of $\mathbf{L}_\Diamond$ depends on the distance between the relative and the extracted expression, a rule associated to this kind of structure must use some auxiliary recursive function on contexts.

Let ζ be the function recursively defined as follows:

$$\begin{aligned}\zeta(\Gamma,\ \Delta,\ i)\ =\ &\texttt{if } \Gamma \texttt{ has the form } (\Gamma_1\,,\,\Gamma_2)_i\\ &\texttt{then } (\Gamma_1\,,\,\zeta(\Gamma_2,\ \Delta,\ i))_i\\ &\texttt{else } (\Gamma\,,\,\Delta)_i\end{aligned}$$

This function takes two contexts Γ and Δ as arguments and also a composition mode i, it returns a context obtained by placing the context Δ as a right sister of the rightest branch of the context Γ which is dominated by the structural connective $(\ldots,\ldots)_i$.

One can derive the following rule, by analyzing ζ's behavior:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{(file : (np\backslash_a s)/_c np\,,\,\_ : ((np\backslash_a s)\backslash_a(np\backslash_a s))/_c np)_a \vdash (np\backslash_a s)/_c np}\,S}{\vdots}}{(John\,,\,(will\,,\,((file\,,\,(without\,,\,reading)_a)_a\,,\,\_ : np)_c)_a)_a \vdash_{\boldsymbol{R}} s}\,\backslash_{\mathbf{E}}}{((John\,,\,(will\,,\,(file\,,\,(without\,,\,reading)_a)_a)_a\,,\,\_ : np)_c \vdash_{\boldsymbol{R}} s}\,(\mathbf{MA}\ c\ a)*}{(John\,,\,(will\,,\,(file\,,\,(without\,,\,reading)_a)_a)_a)_a \vdash_{\boldsymbol{R}} s/_c np}\,/_{\mathbf{I}}}{(which\,,\,(John\,,\,(will\,,\,(file\,,\,(without\,,\,reading)_a)_a)_a)_a)_a \vdash_{\boldsymbol{R}} n\backslash_a n}\,/_{\mathbf{E}}$$

Figure 2.17: Derivation using S combinator

$$\frac{\mathbf{L}_{\Diamond}(i,j) \in R \quad \zeta(\Gamma, < \Delta >_j, i) \vdash_{\boldsymbol{R}} C}{(\Gamma\ , < \Delta >_j)_i \vdash_{\boldsymbol{R}} C}\ \text{Unbound dep}$$

Using this new rule, we can easily deduce a derivation (Fig 2.20) of the sentence (iii) above given the lexicon in the Figure 2.18.

whom	John , Mary	thinks, believes	the	girl
$(n\backslash_a n)/_a(s/_a\Diamond_c \Box_c np)$	np	$(np\backslash_a s)/_a s$	$np/_a n$	n

Figure 2.18: Lexicon for figure 2.20

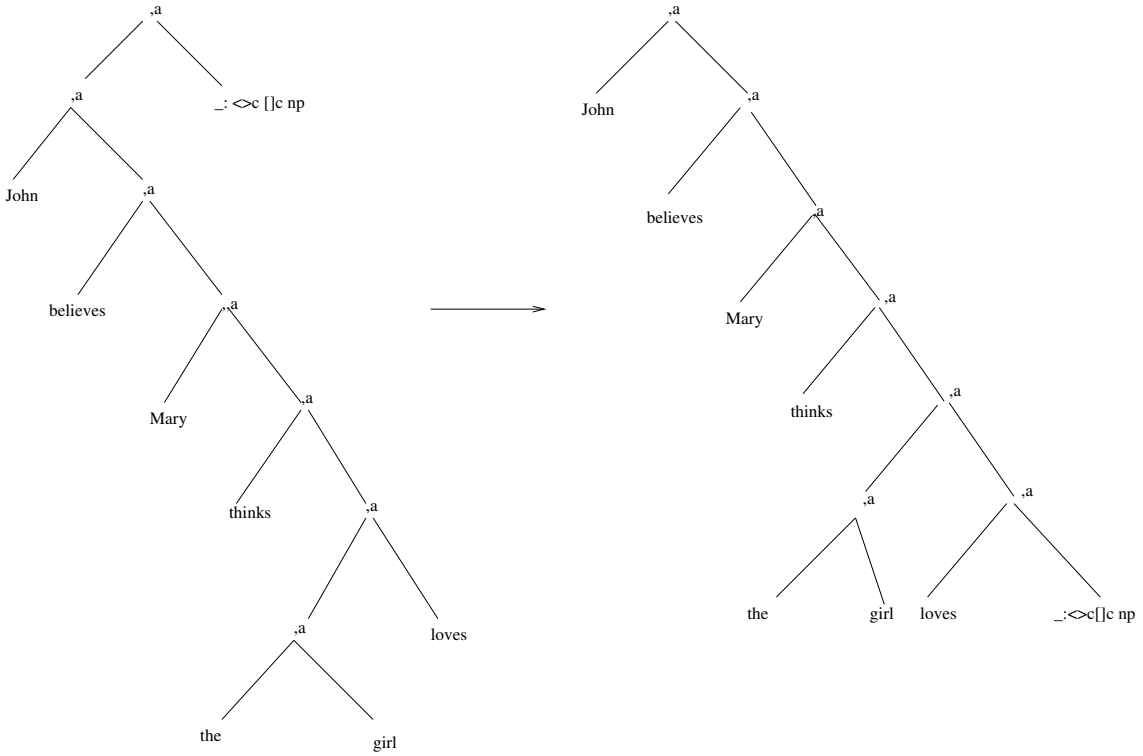

Figure 2.19: Unbound dependencies

$$\dfrac{\dfrac{\dfrac{\dfrac{\ldots.}{(John\,,\,(believes\,,\,(Mary\,,\,(thinks\,,\,((the\,,\,girl)_a\,,\,(loves\,,\,\_:\Diamond_c\Box_c np)_a)_a)_a)_a)_a)_a \vdash s}\ \textit{Unbound dep}}{((John\,,\,(believes\,,\,(Mary\,,\,(thinks\,,\,((the\,,\,girl)_a\,,\,loves)_a)_a)_a)_a\,,\,\_:\Diamond_c\Box_c np)_a \vdash s}\ /_{\mathbf{I}}}{(John\,,\,(believes\,,\,(Mary\,,\,(thinks\,,\,((the\,,\,girl)_a\,,\,loves)_a)_a)_a)_a \vdash s/_a\Diamond_c\Box_c np}\ /_{\mathbf{E}}}{(whom\,,\,(John\,,\,(believes\,,\,(Mary\,,\,(thinks\,,\,((the\,,\,girl)_a\,,\,loves)_a)_a)_a)_a)_a \vdash n\backslash_a n}$$

Figure 2.20: Application of unbound dependencies

2.3 Induction on derivations

2.3.1 Introduction

The preceding techniques — simple application of the natural deduction rules with first-order variables — have a limited range of application. Let us consider for instance three kinds of problems we cannot solve using only these techniques. For solving them, we shall have to consider a powerful reasoning tool: induction on derivations.

2.3.1.1 The rule of isotonicity

Assume we want to now to derive the rule of *isotonicity*:

$$\textbf{if } A \vdash B, \ \textbf{then } A/C \vdash B/C$$

First, let us verify that the techniques used in Figure 2.5 are useless: We consider an arbitrary derivation d_1 of $A \vdash B$, and try to build a derivation of $A/C \vdash B/C$. The only rule which can lead to this sequent is $/_{\mathbf{I}}$. In order to use it, we must build a derivation of $(A/C\ ,\ C) \vdash B$. Unfortunately, the only rule which can lead to this last sequent is $/_{\mathbf{E}}$, which requires a derivation of $A/C \vdash B/C$.

So, we are on a vicious circle. This example shows the limits of our first method, consisting in making simple derivations with type variables.

Nevertheless, isotonicity is derivable in in sequent calculus, even without the cut rule, and in axiomatic Hilbert-like deduction systems. We can also observe derive easily any instance of isotonicity (obtained by replacing A, B and C by some ground formulas.)

For instance, take $A = \texttt{np}$, $B = \texttt{s/(np\s)}$, and $C = \texttt{n}$; the corresponding derivation is shown in Figure 2.21. Notice that this derivation does not contain any subderivation of the sequent $np \vdash s/(np\backslash s)$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{np/n \vdash np/n}\,Ax \quad \dfrac{}{n \vdash n}\,Ax}{np/n, n \vdash np}\,/_{\mathbf{E}} \quad \dfrac{}{np\backslash s \vdash np\backslash s}\,Ax}{(np/n, n), np\backslash s \vdash s}\,\backslash_{\mathbf{E}}}{np/n, n \vdash s/(np\backslash s)}\,/_{\mathbf{I}}}{np/n \vdash (s/(np\backslash s))/n}\,/_{\mathbf{I}}$$

Figure 2.21: An instance of isotonicity

2.3.1.2 A solution: cut rule for natural deduction

Isotonicity will be easy to derive once we can derive a cut rule, similar to the sequent calculus cut rule. This rule is written as follows:

$$\dfrac{\Delta \vdash A \quad \Gamma[A] \vdash B}{\Gamma[\Delta] \vdash B}\,cut$$

Figure 2.22 shows how isotonicity derives from the cut rule. The derivation of this rule is shown in Section 2.3.3.1.

$$\dfrac{\dfrac{\dfrac{\overline{A/C \vdash A/C}\ Ax \quad \overline{A \vdash A}\ Ax}{A/C, C \vdash A}\ /_{\mathrm{E}} \quad \overline{A \vdash B}}{A/C, C \vdash B}\ cut}{A/C \vdash B/C}\ /_{\mathrm{I}}$$

Figure 2.22: Deriving isotonicity from the cut rule

2.3.1.3 Simulation of structural rules

Another similar problem is to derive the following rule:

If $\Gamma \vdash_{\boldsymbol{L1}} A$ **then** $\Gamma \vdash_{\boldsymbol{P} \cup \boldsymbol{L2}} A$.

Although this a conditional rule, one cannot try to build the second derivation with the first one as a subderivation. It seems clear that we should transform each **L1**-step of the first derivation into a sequence of **L2**- and **P**-steps. We need computing tools for doing this transformation.

2.3.1.4 Proofs of non-derivability

It is sometimes useful to verify that some sequent is *not* derivable in our logical system. For instance, let us consider the following sequent, which corresponds to the sentence (*) 'Money is bad master' if we take the small lexicon given in Figure 2.23.

`(np, (np\s)/np, n/n, n)` $\vdash_{\boldsymbol{R}}$ `s`

In order to prove that this sequent is not derivable, one cannot consider the infinite space of possible derivations. It would be better to find an *invariant*, i.e. a property verified by all possible derivations and falsified by the sequent we want to reject. The reader must notice two important facts:

- Once this invariant is defined and proved, the cost of this method is reduced to the cost of checking that the considered sequent falsifies the invariant.

- This is a useful, but incomplete method, which only serves to falsify sequents, not to derive them.

In our example, the invariant is defined by associating a polynomial to every formula or context, the formal variables of which are the atomic types.

Look at (Retoré 2000) for more details on this method.

$$\begin{aligned}
\rho(at) &= at \quad \text{for every atomic type } at \\
\rho(A/B) &= \rho(A) - \rho(B) \\
\rho(A\backslash B) &= \rho(A) - \rho(B) \\
\rho(A \bullet B) &= \rho(A) + \rho(B) \\
\rho(\Box A) &= \rho(A) \\
\rho(\Diamond A) &= \rho(A) \\
\rho((\Gamma, \Gamma')) &= \rho(\Gamma) + \rho(\Gamma') \\
\rho(\langle \Gamma \rangle) &= \rho(\Gamma) \\
\rho(\Gamma \vdash_{\boldsymbol{R}} A) &= \rho(\Gamma) - \rho(A)
\end{aligned}$$

We can prove by induction on derivations, that, if all the rules in R are linear (i.e. each variable appears once in left- and right-hand sides), then ρ maps every derivable sequent $\Gamma \vdash_{\boldsymbol{R}} A$ to the null polynomial.

Since ρ maps our little sequent to the polynomial $n - np$, we conclude that this sequent is not derivable. Notice that, if the system R contains some non-linear rule, like **MC**, the polarity method cannot be applied.

Money	servant, master	is	a	good, bad
np	n	(np\s)/np	np/n	n/n

Figure 2.23: A small lexicon (see 2.3.1.4)

2.3.2 The induction scheme

The three problems presented above can be solved by a powerful scheme: induction on derivations. It allows both to prove properties of derivations and to define recursive functions on derivations.

Let P be any predicate, or, more generally any function[2] defined on derivations. Since these derivations can deal with any sequent $\Gamma \vdash_{\boldsymbol{R}} A$, P has to take in reality three arguments: a context Γ, a formula A, and a derivation d of $\Gamma \vdash A$. Notice that, since d *depends* on Γ and A, the order in which these arguments are given is relevant. If Γ and A can be easily inferred from d, we shall often write $P(d)$ instead of $P(\Gamma,\ A,\ d)$.

This scheme is expressed as follows:[3]

In order to prove the following statement[4]*, :*

$$\forall\, \Gamma\, \forall\, A\, (\forall\, d : \Gamma \vdash_{\boldsymbol{R}} A), P(\Gamma,\ A,\ d)$$

it is enough to prove the following ones:

[2] In effect, predicates are functions which help to build propositions. This is consistent with The Calculus of Constructions, the theoretical foundation of *Coq*.

[3] For sake of clarity, we present this scheme as a logical rule, parameterized by a *predicate* P. The reader must notice that this scheme is easily adaptable for defining functions on derivations. For a detailed presentation of induction and recursion, one can have a look at (Bertot & Castéran 2004).

[4] The third universal quantification must be read as "for any derivation d of $\Gamma \vdash_{\boldsymbol{R}} A$"; in Type Theory, we say that d has type $\Gamma \vdash_{\boldsymbol{R}} A$

Axiom rule: $\forall\, A_1,\ P(Ax(A_1))$.

Rule $/_{\mathbf{I}}$:

$$\forall\, A_1,\, B_1, \Gamma_1,\ (d_1 : (\Gamma_1, B_1) \vdash_{\boldsymbol{R}} A_1),\ P(d_1) \Longrightarrow P(/_{\boldsymbol{I}}(\Gamma_1,\ A_1,\ B_1\, d_1))$$

Rule $/_{\mathbf{E}}$:

$$\begin{array}{c}\forall\, \Gamma_1,\, \Gamma_2,\, A_1,\, A_2, (d_1 : \Gamma_1 \vdash_{\boldsymbol{R}} A_1/A_2),\, d_2 : (\Gamma_2 \vdash_{\boldsymbol{R}} A_2),\\ P(d_1) \wedge P(d_2) \Longrightarrow P(/_{\boldsymbol{E}}(\Gamma_1, A_1, \Gamma_2, A_2, d_1, d_2))\end{array}$$

...

Struct rules:

$$\begin{array}{c}\forall \Gamma_1,\, \Delta\, A_1, r \in R,\, (d_1 : \Gamma_1[r(\Delta)] \vdash_{\boldsymbol{R}} A_1\\ P(d_1) \Longrightarrow P(\boldsymbol{struct}(r, \Gamma_1,\, \Delta, A, d_1))\end{array}$$

In total, 12 conditions must be checked in a construction by induction, and the last one depends on an arbitrary set of structural rules. Moreover, three rules: $\bullet_{\mathbf{E}}$, $\Diamond_{\mathbf{E}}$, and **struct** are applied in subcontexts, as shown by the notation $\Gamma[\,]$; thus their proof often requires a nested induction on the depth of the place at which they are applied. This shows how this kind of proof is done more easily and safely on a computer.

2.3.3 Some constructions by induction

Let us show how the induction scheme helps us in solving the problems presented in Section 2.3.1.

2.3.3.1 Derivation of the cut rule

A first attempt It is tempting to derive the cut rule by an induction with $P(\Gamma,\, B,\, d)$ defined as $\forall\, \Gamma'\, A',\ \Gamma'[A] = \Gamma \Longrightarrow \Gamma'[\Delta] \vdash B$.

Unfortunately, this will not work. In effect, consider a derivation ending by an application of a structural rule like **MC**:

$$\dfrac{\dfrac{d_1}{\Gamma_1 \vdash B}}{\Gamma[\Delta] \vdash B}\ MC$$

The application of **MC** may have duplicated the subterm Δ, and the context Γ_1 may be not of the form $\Gamma'[\Delta]$, where $\Gamma'[A] \vdash B'$. Thus, the induction hypothesis cannot be applied to d_1.

A generalized cut rule It is a very common practice, in the case where some property cannot be proved by a simple induction, to try to prove a generalization of that property. In our case, instead of considering linear contexts, i.e. with a single hole $\Gamma[]$, we generalize the cut rule to contexts with any number of holes. Le us denote by $\Gamma\{\Delta\}$ the replacement of any number of occurrences of Δ in Γ.

So, the rule we shall derive is the following one:

$$\frac{\Delta \vdash A}{\forall \Gamma\, B,\ \Gamma\{A\} \vdash B \Longrightarrow \Gamma\{\Delta\} \vdash B}\ cut'$$

This rule is clearly a generalization of our first cut rule, taking exactly one occurrence of A and Δ. For proving the new rule by induction, we shall define $P(\Gamma,\ B,\ d)$ as $\forall\, \Gamma'\ A',\ \Gamma'\{A\} = \Gamma \Longrightarrow \Gamma'\{\Delta\} \vdash B$.

Sketch of the proof In order to derive **cut'**, we must check the 12 conditions of the induction scheme. The following proof is a proof by induction on the derivation d of $\Gamma\{A\} \vdash B$. Many cases are very similar[5], so we present only the most typical cases, the other ones being left as an exercise.

Axiom rule If the derivation d is an application of the axiom rule, then Γ is the empty context, $B = A$, and $\Gamma\{\Delta\} = \Delta$.

Rule $/_{\mathbf{I}}$ If the last rule of d is $/_{\mathbf{I}}$, then B has the form B_1/B_2, and we have a subderivation d_1 of $\Gamma\{A\}\,,\ B_2 \vdash B_1$. By induction hypothesis, we can build $d'1 : \Gamma\{\Delta\}\,,\ B_2 \vdash B_1$, and, applying $/_{\mathbf{I}}$, we get a derivation of $\Gamma\{\Delta\} \vdash B$.

Rule $/_{\mathbf{E}}$ If d ends by the application of $/_{\mathbf{E}}$, then $\Gamma[A]^*$ can be decomposed into $(\Gamma_1\{A\}\,,\ \Gamma_2\{A\})$, and we have subderivations d_1 of $\Gamma_1\{A\} \vdash B/B_2$ and d_2 of $\Gamma_2\{A\} \vdash B_2$. By induction, we can build derivations $d'1$ of $\Gamma_1\{\Delta\} \vdash B/B_2$ and $d'2$ of $\Gamma_2\{\Delta\} \vdash B_2$. Applying $/_{\mathbf{E}}$ gives us a derivation of $\Gamma\{\Delta\} \vdash B$

Rule $\bullet_{\mathbf{E}}$ If the derivation d ends with an application of $\bullet_{\mathbf{E}}$, then $\Gamma\{A\}$ has the form $\Gamma_1[\Delta_1]$, where

$$\frac{\Delta_1 \vdash A_1 \bullet B_1 \quad \Gamma_1[A_1, B_1] \vdash C}{\Gamma_1[\Delta_1] \vdash C}\ \bullet_{\mathbf{E}}$$

By induction hypothesis, we can build derivations of $\Delta_1\{\Delta\} \vdash A_1 \bullet B_1$ and $(\Gamma_1\{\Delta\})[A_1, B_1] \vdash C$, then, by $\bullet_{\mathbf{E}}$, we have a derivation of $(\Gamma_1\{\Delta\})[\Delta_1\{\Delta\}] \vdash C$. A commutation lemma on substitutions simplifies this sequent into $(\Gamma_1[\Delta_1])\{\Delta\} \vdash C$, which ends this case.

Structural rules This case is very similar to $\bullet_{\mathbf{E}}$, relying on a commutation lemma between the substitution $\Gamma\{\Delta\}$ and the application of a weak Sahlqvist rule. Notice that if some rewrite rule is not of that form, then the commutation lemma is false, and the cut rule cannot be derived.

Remark In fact, the derivation of the cut rule is a recursive transformation which maps any derivation using **cut** into a cut-free derivation. For instance, Figure 2.24 shows a derivation using cut rule. Removing its cut gives us the derivation shown in Figure 2.21.

[5]The proof script (i.e. the sequence of commands the user has to type to prove interactively some theorem) is quite long and repetitive, but many parts of it were copied and pasted

$$\dfrac{\dfrac{\dfrac{\dfrac{}{np/n \vdash np/n}\ Ax \quad \dfrac{}{n \vdash n}\ Ax}{np/n, n \vdash np}\ /_{\mathrm{E}} \quad \dfrac{}{np \vdash s/(np\backslash s)}\ lifting}{np/n, n \vdash s/(np\backslash s)}\ cut}{np/n \vdash (s/(np\backslash s))/n}\ /_{\mathrm{I}}$$

Figure 2.24: an instance of isotonicity (using cut rule)

2.3.3.2 Simulation rule

The problem presented in Section 2.3.1.3 can be solved by an induction on derivations d of $\Gamma \vdash_{\boldsymbol{L1}} A$, instantiating $P(\Gamma, A, d)$ as $\Gamma \vdash_{\boldsymbol{P} \cup \boldsymbol{L2}} A$. This construction is left to the reader, who will notice that, on the 12 cases of the induction scheme, only the last one uses explicitly **L1**, **L2** and **P**. Although the 11 first cases are very simple to solve (less than half a line in*Coq*), it is desirable to get a more generic rule, which we need to prove only once for multiple uses.

Let us denote by $\Gamma \underset{R}{\longrightarrow} \Gamma'$ the relationship "Γ' is obtained by applying a rule $r \in R$ to some subterm of Γ and by $\underset{R}{\longrightarrow}^*$ the reflexive and transitive closure of $\underset{R}{\longrightarrow}$; one can prove by induction the following rule:

$$\dfrac{\Gamma \vdash_{\boldsymbol{R}} A \quad \forall \Delta\ \Delta' (r \in R), r(\Delta) = \Delta' \Longrightarrow \Delta \underset{R'}{\longrightarrow}^* \Delta}{\Gamma \vdash_{\boldsymbol{R'}} A}\ sim$$

To apply this rule, it is enough to verify that any rule in R can be simulated by a sequence of applications of rules in R'.

2.3.3.3 Polarity rule

The proof of the polarity lemma is left as an exercise.

2.3.4 Soundness and completeness of natural deduction

The newly derived cut rule is of a great help to interface natural deduction with two other formal systems: axiomatic presentation *a la Hilbert*, and sequent calculus. Each derivation using one of the three formal systems is translated into the two others. The axiomatic presentation is at the basis of the proofs of soundness and completeness wrt. Kripke models (Kurtonina 1995) The preceding translations extend this property to sequent calculus and natural deduction. Notice that these proofs are constructive: one can use them to remove the cuts from a derivation, or to convert a sequent proof into a natural deduction, etc.

CHAPTER 3

A FIRST LOOK AT *Coq*

Coq(The Coq development team 2004, Bertot & Castéran 2004), is a system designed to develop mathematical proofs, to write formal specifications and to verify that programs are correct with respect to their specification. Using the so-called *Curry-Howard isomorphism*, programs, properties and proofs are formalized in the same language called *Calculus of Inductive Constructions*, which is a λ-calculus with a rich type system. The main originality of *Coq* is that all logical judgments are typing judgments. That implies that, in order to verify that one can trust the proofs made with this system, one has only to verify the type checking algorithm, which is a very little part of *Coq*.

3.1 Terms and types

We do not plan to give here a complete presentation of the Calculus of Inductive Constructions. Our ambition is to give some idea of its expressive power, taking our examples as most as possible in the domain of categorial type grammars.

Let us denote by $t : T$ the judgment " t is a well typed term of type T ". Formally, one should precise an *environment* E and a *context* Γ composed of the declarations of the identifiers which possibly occur in t and T, and the full notation should be $E[\Gamma] \vdash t : T$. For sake of legibility, we shall omit these components of judgments in these notes[1].

Notice that in the Calculus of Constructions, every type is also a term, and — as a term — it has a type too. The type of a type is always an identifier called a *sort*.

In these notes, we shall use mainly the *sorts* `Set`, which is a type for all data types, and `Prop`, which is a type for all logical propositions. Both `Prop` and `Set` have the sort[2] `Type`. In fact, many terms and types combine elements of sorts `Prop` and `Set`, allowing to write program specifications and realizations.

[1] Among the reasons of this omission is the presence of a notion of contexts in categorial grammars, and the possibility of confusion

[2] There is no reason to stop here. The type of `Type` is the sort (also called an *universe*) `Type(1)`, and so on ...

3.1.1 A bestiary of typed terms

Let us present a series of commented typed judgments. The first ones are very similar to type judgments in programming languages, but we shall very soon see the great expressive power obtained using dependent and higher-order types.

3.1.1.1 Numbers

Our simplest example is " `3:nat` ", where `nat` is the type associated to natural numbers. *Coq* libraries define also integers and real numbers, but we shall not use them in this presentation.

3.1.1.2 Functions

If T and T' are types, then $T \to T'$ is the type of *total* functions from T to T'. The term " `fun` $x : T \Rightarrow$ `t'` " has type $T \to T'$ if t' has type T' whenever x has type T. It corresponds to the classical notation $\lambda x : T.\, t'$.

The $\to$ operator is right associative, so the type $T \to T' \to T''$ must be read as $T \to (T' \to T'')$. For instance the addition on natural numbers, bound to the identifier `plus`[3] has type `nat→nat→nat`, and the function " `fun n:nat ⇒ n + n` " the type `nat→nat`.

The application of a function $f : T \to T'$ to an argument $x : T$ is a term written " $f\ x$ " and has type T'. Function application is left-associative, so the application " $f\ x\ y$ " is an abbreviation for " $(f\ x)\ y$ ".

3.1.1.3 Inductive types

Many types used in *Coq* developments are *inductive types*, i.e. types defined by an enumerations of *constructors*. Every value in an inductive type I is assumed to be built through a finite number of applications of the constructors of I.Very simple examples are the type of boolean values, with two constructors, and the type of natural numbers with one recursive constructor, allowing to build an infinity of terms of type `nat` : `O`, `S O`, `S (S O)`, `S (S (S O))`, etc.[4]

```
Inductive bool : Set := true | false.

Inductive nat : Set :=
| O : nat  (* zero *)
| S : nat -> nat. (* successor *)
```

The *Coq* system associates to each inductive definition a possibility of function construction and proof by induction, as well as some reduction rules allowing computation on the associated type.

Moreover, any function defined on an inductive type can be defined by pattern matching, i.e. by giving its value on each constructor.

[3] Like in many functional languages, functions are "curried"; moreover, the infix notation "n+p" is a shorthand for "plus n p".

[4] Happily, the *Coq* system reads and prints these terms as 0, 1, 2, 3, etc.

For instance, the elementary operations on natural numbers can be easily defined, using pattern matching and recursion[5] in a *ML*-like style. as follows:

```
Fixpoint plus (n p : nat) {struct n} : nat :=
  match n with
  |  O => p
  |  S n' => S (plus n' p)
  end.

Fixpoint mult (n p : nat) {struct n} : nat :=
  match n with
  | O => O
  | S n' => plus p (mult n' p))
  end.

Eval compute in (mult 6 6).
36:nat
```

Notice that inductive types may be parameterized, for instance with other types. Here is the definition of a type for binary trees the leaves of which are labeled with some type A.

```
Inductive tree (A:Set):Set :=
| leaf : A -> tree A
| bin : tree A -> tree A -> tree A.
```

It is therefore possible to consider types for trees labeled with natural numbers: " `tree nat` ", or function on integers : " `tree (Z→Z)`, etc.

Another useful inductive type allows us to build terms which denote either a value of type A or no value at all. This is very useful to define partial functions: a partial function f from A into B may be represented by a total function which maps x to `Some` y if $f(x) = y$ and to `None` if x is not in the domain of f.

```
Inductive option (A:Set):Set :=
  Some : A -> option A
| None : option A.

Definition left_son (A:Set) (t: tree A): option (tree A) :=
  match t with
  | (bin  t1 _) => Some t1
  | _           => None
  end.
```

3.1.1.4 Dependent products

The dependent product is a type construct which generalizes the arrow $A \to B$. It allows us to define functions where the result type depends on the value of the argument.

[5]Recall that in the Calculus of Constructions, every considered function is total, thus every recursive definition must have one strictly decreasing argument wrt. a well-founded order. In this case, the `{struct n}` annotation tells the system that `n` is this argument, considering the order associated to the subterm relationship.

For instance, the function `left_son` above has a result type which can be `option(tree T)` for any data type T. In the Calculus of Constructions, this function has the following type :

$$\forall A : Set, tree\ A \rightarrow option(tree\ A)$$

This new construct, which can express much more than the simple polymorphism, is called the *dependent product*. Many other examples will be given in the following pages.

3.1.1.5 Dependent types

A *dependent type* is a type whose expression contain arbitrary terms. For instance, " a vector of size n ", " a prime number greater than n " are such types. Dependent types are very useful, for instance to build specifications.

Here is the declaration of a polymorphic type of vectors : For any data type A, `null_vector` is a vector of size 0, and for any natural number n, value $a : A$ and vector v of size n, then `vector_cons A n v` is a vector of size $n + 1$.

```
Inductive vector (A:Set): nat -> Set :=
| null_vector : vector A 0
| cons_vector : ∀(n:nat), A -> vector A n -> vector A (S n).
```

3.2 Propositions and proofs

A *proposition* is any type of sort `Prop`. We shall see that all techniques used in the framework of types for programming languages have their logical counterpart : inductive, higher-order, dependent types. What about the notion of truth of a proposition?

Coq follows the approach proposed by Heyting (Heyting 1971), which replaces the question "is the proposition P true?" with the question "what are the proofs of P (if any)?". Proofs are actually represented by terms of the typed λ-calculus. For instance, a proof $P \Rightarrow Q$ is a function that given an arbitrary proof of P *constructs* a proof of Q.

The so-called *Curry–Howard isomorphism* establishes a correspondence between functional programming and natural deduction, see for instance Girard, Lafont, and Taylor (Girard, Lafont & Taylor 1989) and the papers of Curry and Feys (Curry & Feys 1958) and Howard (Howard 1980). From a practical point of view, this correspondence allows us to use the same ideas and tools for both programming and reasoning.

Using dependent types, it is easy to characterize a *predicate* on type T as a function of type $T \rightarrow$`Prop`. For instance, the predicate $<$ is a function of type `nat`$\rightarrow$ `nat`$\rightarrow$ `nat` and "being greater than 36" is the abstraction " `fun n:nat` $\Rightarrow$ `36 < n` ".

3.2.0.6 Remark

The reader must not believe that the world of types and programs and the world of propositions and proofs are totally disjoint. In fact, predicates and

program specifications, as well as proofs and certified programs are in general made from blends of types and terms of both sorts `Set` and `Prop`.

3.3 Examples

Many propositions and connectives are defined as inductive type of sort `Prop`. For instance, the (intuitionistic) disjunction is an inductive type with two constructors, which represent the two introduction rules for disjunction. As for the other connectives, the elimination rule is automatically generated by *Coq*.

```
Inductive or (A : Prop) (B : Prop) : Prop :=
    or_introl : A -> A \/ B
  | or_intror : B -> A \/ B.
```

Universal quantification (first and higher-order) is represented with dependent products. For instance, the following theorem — the automatically generated induction principle for `nat` — contains quantification both on natural numbers and predicates.

```
∀ P : nat -> Prop,
  P 0 -> (∀ n : nat, P n -> P (S n)) ->
  ∀ n : nat, P n
```

Notice that inductively defined propositions can be seen as a generalization of *Prolog* definitions: each constructor is associated with a clause. For instance, Figure 3.1 shows the definition of the $\leq$ total order on natural numbers. As for inductive types like `nat`, *Coq* associates to each inductively define predicate an induction scheme. In the case of $\leq$, the system automatically generates the scheme shown in Figure 3.2[6].

```
X <= X.
X <= s Y :- X <= Y

Inductive le (n:nat): nat-> Prop :=
| le_n : n <= n
| ∀ m:nat, n <= m -> n <= S m.
```

Figure 3.1: Prolog and Coq definitions of $\leq$

$$\frac{n \leq p \quad P(n) \quad \forall q\, n \leq q \Longrightarrow P(q) \Longrightarrow P(S(q))}{P(p)}\ \texttt{le\_ind}$$

Figure 3.2: Induction scheme on $\leq$

[6]For sake of legibility, we removed the universal quantifiers on P, n and p.

3.4 Tactics and Automation

Since *Coq* can express higher-order logic, one cannot expect fully automated proofs of theorems. On the opposite side, since proofs are represented as huge λ-terms, on cannot expect the user to type these proofs by hand. As in many proof assistants following *LCF* (Gordon, Milner & Wadsworth 1979), proving a theorem or realizing some specification is performed to *tactics* and *tacticals*, which help the user to build terms of a given type through a divide-and-conquer strategy.

Le us call a *goal* the problem of searching a term of a given type T. A *solution* of this goal is a term t of type T. Applying a tactic consists in replacing a goal T by a finite sequence of *subgoals* $T_1, \ldots, T_n$, whose solution $t_1, \ldots, t_n$ must contribute to the building of a solution t of T.

Tactics have a very large range of possibilities, from the tactics which implement basic logical rules (introduction, elimination) to decision procedures (quantifier-free Presburger Arithmetic, simplification of expressions on rings and fields, semi-decision of first-order logic, etc.)

On some domains, tactics are not expected to be complete: they may fail when called outside their domain of application, but they cannot help you to build "false" proofs.

Moreover, *Coq* comes with a programming language for tactics, called `Ltac`, which allows the developer to build specialized tactics for the considered domain. Thus, we replace an impossible "fully automatic deduction" with "programmable assisted deduction".

CHAPTER 4

CATEGORIAL GRAMMARS IN COQ

In this section, we show how which features of the Calculus of Constructions and the *Coq* system were used to build a logical workshop around categorial grammars. First, we show how grammars are represented using inductive and dependent types, then how computations and tactics help us to prove and use generic results.

4.1 Representing modes and atomic types

In a given categorial grammar, there are in general few different modes and atomic types. It is then easy to represented them as enumerated data types, Here is a small definition with four distinct atomic types:

```
Inductive atoms : Set :=
| np  | n  | vp | s.
```

4.2 Representing formulas and contexts

Recall that type formulas and contexts are defined with respect to three parameters: a set I of binary modes, a set J of unary modes, and a set A of atomic types. Figure 4.1 shows the definition of the type of formulas.

For instance, the constructor `At` specifies that, if a is an atomic formula (i.e. a term of type `A`), then the term "`At a`" is a formula (i.e. a term of type `Form`.) In the same way, if i is a binary mode, A and B formulas, then the term "`Slash i A B`" is the formula we write usually $A/_i B$.

Notice that this declaration takes place in an environment where the type variables `I`, `J` and `A` have been already declared. *Coq* is provided with a mechanism which automatically generalizes these three parameters. Thus, `Form` becomes a type operator depending on modes and atomic types. For instance if `binaries`, `unaries` and `atoms` are three data types, then the type for corresponding formulas is the application "`Form binaries unaries atoms`".

```
Inductive Form   : Set :=
| At : A -> Form
| Slash : I -> Form  -> Form  -> Form
| Backslash : I -> Form -> Form  -> Form
| Dot : I -> Form  -> Form  -> Form
| Box : J -> Form  -> Form
| Diamond : J -> Form  -> Form .
```

Figure 4.1: Inductive type for formulas

The inductive definition of contexts is very similar to the definition of formulas: there is a constructor for contexts with a single formula, a binary constructor associated with the $(\dots,\dots)_i$ construct, and one for the $\langle\dots\rangle^j$ construct.

Let us consider rules like $\bullet_{\mathbf{E}}$, $\Diamond_{\mathbf{E}}$, **struct**, and **cut**, which all use decomposition of contexts under the form $\Gamma[\Delta]$. In our implementation, we consider such a decomposition as the filling of a *linear context* $\Gamma[]$, i.e. a context with one hole (represented here as $[]$) by some context Δ. Linear contexts are represented as Huet's *zippers* (Huet 1997, Huet 2003). Zippers are a very efficient data structure representing a tree and a pointer to one of its subtrees. They are compatible with purely functional programming, and used in several libraries, including in the field of computational linguistics (Huet 2003, Ranta 2004).

Figure 4.2 shows how to represent a context with one hole as a zipper, which is more or less the same tree, put upside down, the zipper's root being the hole of the linear context. Figure 4.3 shows the type definition for these structures and the filling function.

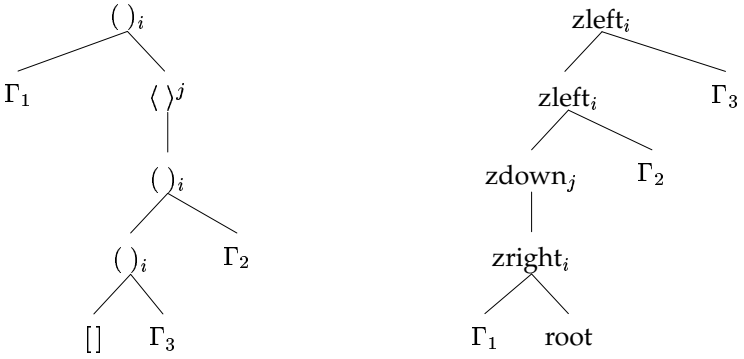

Figure 4.2: A linear context and its corresponding zipper

Lexicons are defined as functions mapping words to lists of formulas, and resources are structures composed of typed words (hypothetical or taken from the lexicon). Thus, inductive data types are enough to describe multimodal grammars and structured sentences.

```
(* linear contexts  (zippers) *)
Inductive zcontext : Set :=
   | zroot : zcontext
   | zleft  : I ->  zcontext -> context -> zcontext
   | zright : I -> context->  zcontext -> zcontext
   | zdown : J ->  zcontext -> zcontext.

(* fill a linear context with a context *)

Fixpoint zfill (z:zcontext)(Gamma : context) struct z
   : context :=
   match z with zroot => Gamma
               | zleft i z1 Gamma2 =>
                    zfill z1 (Comma  i Gamma Gamma2)
               | zright i Gamma1 z2 =>
                    zfill z2 (Comma i Gamma1 Gamma)
               | zdown j z1 => zfill z1 (TDiamond j Gamma)
           end.
```

Figure 4.3: Code for linear contexts

4.2.1 Representation of derivations

The representation of derivations in categorial grammars uses inductive dependent types. We show how to define the rules for natural deduction, but sequent calculus has a very similar definition.

For which reason do we use dependent types? A derivation of $\Gamma \vdash_{\boldsymbol{R}} A$ can be seen as a tree whose notes are labeled by rule names (see Table 2.1), and it is natural to define an inductive type called `natded` with a constructor for each rule. Dependent types are needed to express the constraints between derivations: for instance, the constructor `SlashI` takes as main arguments a context Γ, two formulas F and G, a binary mode i, a term d_1 of type " `natded` $(\Gamma\ ,\ G)_i\ \ F$ " and returns a term of type " `natded` $\Gamma\ \ F/_iG$ ". If d_1 had another type (for instance " `natded` $\Gamma\ \ G\ \ F$ ") the constructor would be impossible to apply. Figure 4.4 show an extract of `natded`'s type definition[1].

4.2.2 Representation of derived rules

All derived rules are represented as functions (with dependent types) which build derivations. Universal quantifiers are used to bind atom types, modes, formula types, contexts, sets of structural rules, etc. Constraints on structural rules are expressed as preconditions.

For instance, Figure 4.5 shows how the Geach and cut rules can be represented as universally quantified statements.

[1]For sake of legibility, we have erased some arguments from the real *Coq* implementation.

```
Inductive natded : context A W -> Form A -> Type :=
  | Wd  : ∀ r  F, natded  F F
  | SlashI :
      ∀ (Gamma:context A W) (F G:Form A) (i:I),
        natded (Comma i Gamma (form G)) F ->
        natded Gamma (Slash i F G)
  | BackI :
      ∀ (Gamma:context A W) (F G:Form A) (i:I),
        natded (Comma i (form G) Gamma) F ->
        natded Gamma (Backslash i G F)
  | DotI :
      ∀ (Gamma Delta:context A W) (F G:Form A) (i:I),
        natded Gamma F ->
        natded Delta G ->
        natded (Comma i Gamma Delta) (Dot i F G)
  | DiamI :
      ∀ (Gamma:context A W) (F:Form A) (j:J),
        natded Gamma F ->
        natded (TDiamond j Gamma) (Diamond j F)
  | BoxI :
      ∀ (Gamma:context A W) (F:Form A) (i:J),
        natded (TDiamond i Gamma) F ->
        natded Gamma (Box i F)
  ...
```

Figure 4.4: A type definition for natural deduction (simplified)

```
∀(I J A : Set)
  (R : extension I J) (a : I),
in_extension (L1 a) e ->
∀ F G H : Form I J A,
natded  R (form (H \a G) ((H \a F) /a G \a F)

∀ (I J A : Set) (R : extension I J)
  (ZGamma : zcontext I J A ) (Delta : context I J A )
  (F G : Form I J A),
weak_sahlqvist_ext R ->
natded decI decJ R Delta F ->
natded  R (zfill ZGamma (form F)) G ->
natded  R (zfill ZGamma Delta) G
```

Figure 4.5: Representation in *Coq* of some derived rules (simplified)

4.3 Specialized tools

The *Coq* system already provides us with a lot of general purpose tactics : introduction and elimination of connectives and quantifiers, automatic generation

of induction schemes for inductive types, etc. For instance, the constructions of section 2.3 use the induction scheme for type `natded`.

Besides these tools, it is necessary to provide the user with some specialized tools for the considered domain.

4.3.1 Navigating in contexts

Recall that some basic or derived rules are expressed in terms of linear contexts, e.g. $\bullet_{\mathbf{E}}$, $\Diamond_{\mathbf{E}}$, **struct**, **cut**. In other terms, solving a goal of the form $\Gamma \vdash A$ needs to express the context Γ under the form $\Gamma_0[\Delta]$.

The type definition for zippers, together with the filling function, is such that both forms are *convertible*, hence considered as identical. It is then easy to build tactics for moving the focus upside, downside, etc, by converting some decomposition $\Gamma[\Delta]$ into another one which is convertible.

4.3.2 Tactics associated with deduction rules

To each constructor for natural deduction, we associate some tactic which call the corresponding constructor with the correct arguments. For instance calling the tactic `dotI` on a goal of the form "`natded` R $(\Gamma_1, \Gamma_2)_i$ $F \bullet_i G$" generates the goals "`natded` R Γ_1 F" and "`natded` R Γ_2 G".

For applying derived rules, one can either use *Coq*'s basic tactics `apply`, `elim`, or use `Ltac` to automatize the verification of some preconditions. For instance, an attempt to use the polarity rule must trigger an automatic verification that all structural rules are linear.

For the rules which use linear contexts, if the current context is already decomposed under the form $\Gamma[\Delta]$, this decomposition is used by the tactic. If such decomposition is missing, an additional subgoal is generated, in order to determine to which subcontext the rule must be applied. Using linear contexts to point where some rule has to be applied comes directly from the "proof by pointing" technology (Bertot, Kahn & Théry 1994).

4.3.3 Proofs and computations

Let us consider again the polarity rule presented in Section 2.3.1.4. It is possible to build a tactic for proving that some sequent $\Gamma \vdash A$ is not derivable. This tactics can proceed as follows: associating a polynomial to Γ and A, simplifying this polynomial, comparing the result with 0, and finally applying the rule. We notice that, once the rule is proved and stored, the cost of this method is that of a decision procedure on integer arithmetics, mainly based on computation than deduction. This decision procedure is implemented using the tactic `ring`, written by Boutin and Loiseleur, which uses the *proof by reflection* technique (Harrison 1995, Boutin 1997, Bertot & Castéran 2004).

4.4 Evolution of *Icharate*

A first version of *Icharate* was made in 2003 (Anoun & Castéran 2004), restricted to Lambek calculus. Equivalence between axiomatic presentation, sequent calculus and natural deduction, as well as soundness and completeness

are proved for **NL**, **L**, **NLP** and **LP**. The current version is an extension to the multimodal system. Moreover, it contains new features, like specialized tactics, and a semantic interface (Montague-like semantics) is in construction.

At this date[2], the system is in constant evolution. New tactics are added, their interest is measured by the simplification they induce on a corpus of examples.

The main weakness of the present version is the user interface, which is by now not friendly at all. Tools like *Pcoq* (Lemme project (Inria) 2004) will help us to provide a graphical interface to the deduction system.

Another direction of evolution is to open *Icharate* to the plug-in of external tools, like *Grail*, for instance. Our system is made for investigating the properties of generic schemes of derivation, but cannot be as efficient as other tools, as far as analysis of real sentences. We plan to implement tactics for calling external tools, in order to get the analysis and/or semantic term of some sentences. Our system will either accept these results as axioms, or, if the external tool gives enough information, use this data for building, then following a proof plan, so that no trust in the external tool is needed.

Other planned evolutions will concern a more powerful semantic interface, not restricted to the simply typed λ-calculus, for taking into account dynamics aspects of semantics, and to implement other formalisms for computational linguistics.

[2]June 15, 2004

CONCLUSION

We have introduced two kinds of tools for reasoning inside and about such a rich and complex formalism as multimodal categorial grammars: Grail for writing and testing grammar fragments and Icharate for studying more generic schemes, at the price of some loss of automaticity.

Putting these tools together will help us to build a powerful toolkit for research on categorial grammars.

BIBLIOGRAPHY

Ajdukiewicz, K. (1935), 'Die syntaktische Konnexität', *Studies in Philosophy* **1**, 1–27.

Anoun, H. & Castéran, P. (2004), Lambek calculus in Coq, Technical report, Contributions to the Coq System.

Bar-Hillel, Y. (1964), *Language and Information. Selected Essays on their Theory and Application*, Addison-Wesley, New York.

Bertot, Y. & Castéran, P. (2004), *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*, Texts in Theoretical Computer Science. An EATCS series, Springer Verlag.

Bertot, Y., Kahn, G. & Théry, L. (1994), Proof by pointing, *in* 'Symposium on Theoretical Aspects Computer Software', number 789 *in* 'Lecture Notes in Computer Science', Springer Verlag.

Boutin, S. (1997), Using reflection to build efficient and certified decision procedures, *in* 'Theoretical Aspects of Computer Science', Vol. 1281 of *Lecture Notes in Computer Science*, Springer-Verlag.

Buszkowski, W. & Penn, G. (1990), 'Categorial grammars determined from linguistic data by unification', *Studia Logica* **49**, 431–454.

Carpenter, B. (1998), *Type-logical Semantics*, MIT Press, Cambridge, Massachusetts.

Curry, H. B. & Feys, R. (1958), *Combinatory Logic I*, North- Holland.

Girard, J.-Y., Lafont, Y. & Taylor, P. (1989), *Proofs and types*, Cambridge University Press.

Gordon, M., Milner, R. & Wadsworth, C. (1979), *Edinburgh LCF: A mechanized logic of computation*, Vol. 78 of *Lecture Notes in Computer Science*, Springer-Verlag.

Harrison, J. (1995), Meta theory and reflection in theorem proving: a survey and critique, Technical report, SRI International Cambridge Computer Science Research Center.

Hepple, M. (1990), The Grammar and Processing of Order and Dependency, PhD thesis, Edinburgh.

Heylen, D. (1999), Types and Sorts: Resource Logic for Feature Checking, PhD thesis, Utrecht Institute of Linguistics OTS, Utrecht University.

Heyting, A. (1971), *Intuitionism - an Introduction*, North-Holland.

Howard, W. A. (1980), The formulae-as-types notion of construction, *in* J. P. Seldin & J. R. Hindley, eds, 'To H. B. Curry: Essays on combinatory logic, Lambda Calculus and Formalism', Academic Press, pp. 479–490.

Huet, G. (1997), 'Functional pearl: The zipper', *Journal of Functional Programming* **7**(5), 549–554.

Huet, G. (2003), Zen and the art of symbolic computing: Light and fast applicative algorithms for computational linguistics, *in* 'Practical Aspects of Declarative Languages, 5th International Symposium', number 2562 *in* 'Lecture Notes in Computer Science'.

Kurtonina, N. (1995), Frames and Labels, PhD thesis, University of Utrecht.

Lambek, J. (1958), 'The mathematics of sentence structure', *American Mathematical Monthly* **65**, 154–170.

Lambek, J. (1961), On the calculus of syntactic types, *in* R. Jacobson, ed., 'Structure of Language and its Mathematical Aspects, Proceedings of the Symposia in Applied Mathematics', Vol. XII, American Mathematical Society, pp. 166–178.

Lemme project (Inria) (2004), 'Pcoq: A graphical user-interface for coq'. http://www-sop.inria.fr/lemme/pcoq/index.html.

The Coq development team (2004), *The Coq reference manual*, LogiCal Project. Version 8.0.
URL: *http://coq.inria.fr*

Moortgat, M. (1997), Categorial type logics, *in* J. van Benthem & A. ter Meulen, eds, 'Handbook of Logic and Language', Elsevier/MIT Press, chapter 2, pp. 93–177.

Moot, R. (1998), Grail: an automated proof assistant for categorial grammar logics, *in* R. Backhouse, ed., 'Proceedings of Calculemus/User Interfaces for Theorem Provers', pp. 120–129.

Moot, R. & Puite, Q. (2002), 'Proof nets for the multimodal Lambek calculus', *Studia Logica* **71**(3), 415–442.

Morrill, G. (1994), *Type Logical Grammar*, Kluwer Academic Publishers, Dordrecht.

Ranta, A. (2004), 'Grammatical framework'. Home page: `http://www.cs.chalmers.se/~aarne/GF/`.

Retoré, C. (2000), 'The logic of categorial grammars', Notes of ESSLLI. Last version: http://www.labri.fr/Recherche/LLA/signes.

Steedman, M. (1996), *Surface Structure and Interpretation*, The MIT Press.

van Benthem, J. (1995), *Language in Action: Categories, Lambdas and Dynamic Logic*, MIT Press, Cambridge, Massachusetts.