HAL
open science

On the use of advanced logic programming languages

Solange Coupet-Grimal, Olivier Ridoux

To cite this version:

Solange Coupet-Grimal, Olivier Ridoux. On the use of advanced logic programming languages. [Research Report] RR-2391, INRIA. 1994. inria-00074284

HAL Id: inria-00074284

https://inria.hal.science/inria-00074284v1

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

ON THE USE OF ADVANCED LOGIC PROGRAMMING LANGUAGES IN COMPUTATIONAL LINGUISTICS

Solange COUPET-GRIMAL et Olivier RIDOUX

N° 2391

Octobre 1994

PROGRAMME 2

Rapport de recherche

ISSN 0249-6399

ON THE USE OF ADVANCED LOGIC PROGRAMMING LANGUAGES IN COMPUTATIONAL LINGUISTICS

Solange COUPET-GRIMAL* et Olivier RIDOUX**

Programme 2 — Calcul symbolique, programmation et génie logiciel
Projet Lande

Rapport de recherche n° 2391 — Octobre 1994 — 31 pages

Abstract: Computational Linguistics and Logic Programming have strong connections, but the former uses concepts that are absent from the most familiar implementations of the latter. We advocate that a Logic Programming language needs not feature the Computational Linguistics concepts exactly, it must only provide a logical way of dealing with them. We focus on the manipulation of higher-order terms and the logical handling of context, and we show that the advanced features of Prolog II and λProlog are useful for dealing with these concepts. Higher-order terms are native in λProlog, and Prolog II's infinite trees provide a handy data-structure for manipulating them. The formulas language of λProlog can be transposed in the Logic Grammar realm to allow for a logical handling of context.

Key-words: Logique Programming, Computational Linguistics, λProlog, Prolog II

(Résumé : tsvp)

Submitted to publication.

*CMI, 39 rue Joliot-Curie, 13453 Marseille Cedex 13, FRANCE. solange@gyptis.univ-mrs.fr
**ridoux@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 84 71 71

Sur l'usage de langages de programmation logique avancés en informatique linguistique

Résumé : L'informatique linguistique et la programmation logique ont des liens très forts, mais la première utilise souvent des concepts qui n'appartiennent pas aux mises en œuvre les plus courantes de la seconde. Nous arguons qu'un langage de programmation logique ne doit pas nécessairement proposer exactement les concepts utilisés en informatique linguistique, mais seulement un moyen logique de les émuler. Nous nous intéressons plus particulièrement à la manipulation des termes d'ordre supérieur et à la prise en compte logique du contexte, et nous montrons que les dispositifs avancés de Prolog II et λProlog sont utiles pour les émuler. Les termes d'ordres supérieur sont indigènes à λProlog, tandis que Prolog II offre une structure de donnée commode pour les émuler. Le langage de formule de λProlog peut être adapté à un formalisme de grammaire logique pour permettre la gestion logique du contexte.

Mots-clé : Programmation logique, informatique linguistique, λProlog, Prolog II

1 INTRODUCTION

Logic Programming has connections with Computational Linguistics at both the syntactic level and the semantic level. Logic Programming, via Prolog, can be considered as a by-product of studies on automating the analysis of natural language [11]. The relationship was refined through the notion of Logic Grammars [48, 1], but eventually natural language formalisms became more sophisticated independently. We are now at a stage in which more sophisticated natural language grammar formalisms like *Generalized Phrase Structure Grammar* (GPSG [22]) have no clear counterpart in Prolog, the standard incarnation of Logic Programming. At the semantic level, we find the same discrepancy between the more sophisticated formalisms like *compositional semantics* [41] and the comparatively rudimentary first-order terms of the Herbrand Universe used in Prolog [56, 36].

However, it is not necessary that these concepts are featured as such in a Logic Programming language. It is only required that the concepts can be emulated conveniently. Indeed, there exists a too broad range of desirable features to incorporate all of them in a programming language. Only a convenient choice of primitive features must be incorporated. A touchstone for convenience in Logic Programming is that the desired features can be emulated logically and yet efficiently using the primitive features. As a counter-example, Prolog can be used for representing and manipulating higher-order terms because it is a universal computation language, but it is often the case that logic is traded for efficiency [47]. So, it seems Prolog is not convenient for representing and manipulating higher-order terms.

More sophisticated incarnations of Logic Programming exist that fit more logically the purpose of programming Computational Linguistics applications. For instance, Prolog II offers a more general form of first-order terms, namely the *rational trees*, and λProlog offers a more general form of terms and formulas, namely the *simply typed λ-terms* and the *hereditary Harrop formulas*[1].

In this paper, we study through examples how Prolog II and λProlog yield new and more appropriate solutions for representing higher-order terms and for handling context. We insist that there is a necessary gap between Logic Programming and Computational Linguistics, but that it is harmless if it can be bridged by a logical encoding of the desired features. A Logic Programming system and a Computational Linguistics system must remain two different things, even if somewhat close.

We have choosen Prolog II and λProlog because they offer capabilities that are useful, but not fully understood in general. Furthermore, the rational trees of Prolog II can be seen as a first step towards the coreference constraints that feature terms domains like the ψ-terms propose. We have also observed that programmers used to Prolog consider λProlog a difficult language. So, we illustrate this paper with many programming examples. We insist on a few points related to the manipulation of λ-terms, the understanding of which may help avoiding gross mistakes that can be seen in beginners' programs, and occur sometimes in print.

This paper is organized into five sections. First, we show why higher-order terms and a logical handling of context are two features that may be desired in the writing of a Computational Linguistics application. We also show that familiar representations of higher-order terms in Prolog are incorrect. Second we present the specific features of Prolog II and λProlog. Third, we present how Prolog II and λProlog contribute to a more correct handling of higher-order terms. Fourth, we present how the extended clause language of λProlog is the basis of a Logic Grammar formalism that provides for a logical handling of context. Finally, we apply our solutions to a natural language query system.

2 MOTIVATIONS

2.1 Representing higher-order terms

2.1.1 Compositional construction of semantic formulas

A number of works on the representation of the semantics of natural language in Logic Programming rely on the three-branches quantifier formalism, first introduced by Colmerauer in [9]. This is a tree representation of the formula that carries the meaning of a sentence. This representation involves quantifiers with three arguments. For instance, the semantics of the sentence "The minister is working" is the formula *exist(x, minister(x), work(x))*. This approach has been used by many researchers (particularly by Dahl [17] and Pique [49]). Saint-Dizier extended this idea in [52]. Alternative formulations were also proposed by McCord [35] and Pereira [46]. Related work can be found in [18], [19], and [6] as well as in the book of Gal, Lapalme, Saint-Dizier, and Somers [21]. The representation we use in this paper is somewhat different since it remains in the framework of the first-order classical

[1] All the emphasized terms will be explained later.

logic. But the sequel can be applied to other formulations as well (like those using three-branches quantifiers). Here, the semantics of the sentence "The minister is working" is the formula $\exists x\ minister(x) \wedge work(x)$.

The semantics of a sentence is defined by syntactic induction. A basic semantic item is associated with each terminal symbol of the grammar. Then, composition rules yield the semantics of a phrase by combining the semantics of its syntactic components. In the example above, the sentence consists of the verb phrase "is working" and the noun phrase "the minister". The latter consists of the article "the" and the noun "minister". Basic semantic structures associated with "minister" and "is working" are $minister(x)$ and $work(x)$ respectively, that correspond to the predicates "to be a minister" and "to work".

The semantic structure associated with "the" is slightly more complex: $\exists x\ y(x) \wedge z(x)$. Informally, the global formula is derived by the composition rule consisting in substituting the variables y and z in the formula above by "minister" and "work". In fact, these formulas can be viewed as functions of their free variables ($x \mapsto minister(x)$, $x \mapsto work(x)$, and $y,z \mapsto \exists x\ y(x) \wedge z(x)$). Hence, because λ-terms can also be interpreted as functions, the λ-calculus offers an ideal framework, both precise and powerful[2].

The three formulas corresponding to "minister", "work", and "the" are respectively the λ-expressions: $\lambda u(minister\ u)$, $\lambda v(work\ v)$, and $\lambda y \lambda z \exists x((y\ x) \wedge (z\ x))$. They are assembled to form the semantics of the whole sentence, and then *β-reduced*. Thus, for the sentence "The minister is working", we have

$$\underline{(\ (\underline{\lambda y} \lambda z \exists x((\underline{y}\ x) \wedge (z\ x))\ \underline{\lambda u(minister\ u)}\ )\ \lambda v(work\ v)\ )}\ ^{3}$$

which will be successively β-reduced into

$$(\ \lambda z \exists x((\ \underline{\lambda u}(minister\ \underline{u})\ \underline{x}\ ) \wedge (z\ x))\ \lambda v(work\ v)\ )\ ,$$

$$\underline{(\ \underline{\lambda z} \exists x((minister\ x) \wedge (\underline{z}\ x))\ \underline{\lambda v(work\ v)}\ )}\ ,$$

$$\exists x((minister\ x) \wedge \underline{(\ \underline{\lambda v}(work\ \underline{v})\ \underline{x}\ )})\ ,\ \text{and}\ \exists x((minister\ x) \wedge (work\ x))\ .$$

The last expression, where all possible reductions have been performed, is called a *normal form*. It can be passed to some semantic evaluator. In general, a sequence of β-reductions may not terminate in a normal form, but if λ-terms obey supplementary conditions such as typing, every sequence of β-reductions terminates.

In concrete Prolog implementations of these semantic structures, λ-calculus variables are generally represented by Prolog variables and β-reductions amount to Prolog unifications. For example, $\lambda u(minister\ u)$ is translated into the Prolog term $abstr(U, minister(U))$, U being a Prolog variable. The main advantage of this approach is the straightforwardness of the implementation. All the substitutions in the λ-calculus are automatically realized by the unification mechanism as follows:

```
beta_reduce( appl( abstr(X, E), F ), E ) :- X = F .          % Caution!
```

However, this method raises several problems that are stressed in Pereira's survey paper [47]. First, it is not in accordance with Prolog semantics. A Prolog variable is just an unknown first-order term, and it cannot be overloaded with connotations such as scope and binding time that are extraneous to first-order terms. Using it for coding a metavariable of the λ-calculus amounts to programming by *side effects*. Second, such an implementation of the λ-calculus is not sound and may produce undesired results mainly because it is careless of the scope of λ-variables. Let us consider, for example, the sentence "Peter and Paul are working". The expected semantics is $work(Peter) \wedge work(Paul)$. However, if the verb phrase semantics is represented in Prolog as $abstr(X, work(X))$ (X being a Prolog variable) then X cannot be unified successfully with two different constants (viz. *Peter* and *Paul*).

The classical Prolog solution to the last problem is to use the predicate *copy_term*. It copies a term and renames its variable in the copy. This is not declarative because it gives a meaning to being still unknown or not, and it breaks the initial straightforwardness because logical variables that are not intended to represent bound λ-variables must not be renamed. In contrast, if the verb phrase semantics is represented by an actual abstraction, $\lambda x(work\ x)$, then there is no problem[4].

[2] The λ-calculus is an equality theory that can model functional computation.

Its terms are built with constants and variables using two rules. If E, F, and x are two terms and a variable, the *abstraction* rule builds term $\lambda x(E)$, and the *application* rule builds term $(E\ F)$. Abstraction can be understood as defining function. It introduces the notion of *free* and *bound* occurrences of variables: occurrences of variable x outside all abstractions $\lambda x(E)$ are free, otherwise they are bound.

The main axiom of the theory is called β: $(\ \lambda x(E)\ F\ ) =_\beta E[x \leftarrow F]$, if free variables of F do not occur bound in E. The intuition of this axiom is the function call, and the consistent substitution of an actual parameter, F, to a formal parameter, x. To *β-reduce* is to use the axiom as a rewriting rule from left to right. Two terms are *β-equivalent* if they β-reduce to a common term.

[3] Parts to be reduced are underlined. Effective arguments and variables they are substituted to are underlined twice.

[4] Of course, this does not mean that using abstraction will solve every problem of coordination.

2.1.2 Quantifiers in semantic formulas

Another problem with object-level variables is to decide the status of quantifier $\exists$ and variable x in $\exists x\ y(x) \wedge z(x)$. The variable is supposedly bound by the quantifier, but it will probably be replaced by some term when the resulting formula is evaluated. Again, to represent it as *exist(X, ...)* where X is a Prolog variable is only a grossly approximate solution. A logical solution is to consider abstraction as a syntactic quantification and encode the semantics of the quantifier by a higher-order function. For instance, $\exists x\ y(x) \wedge z(x)$ is represented by the λ-term $(exists\ \lambda x((y\ x) \wedge (z\ x)))$.

So, higher-order terms handle properly several aspects of object-level variables: their scope and their substitutability. Moreover, we have seen that they are useful for implementing a compositional construction of formulas. Note that the formulas to be encoded by higher-order terms need not be higher-order terms themselves. In the examples of this section, they are first-order predicate formulas. So, we will not insist on the direct availability of higher-order terms, but rather on the possibility to emulate them. Thus, the techniques used will more likely apply to any kind of object-level formulas.

2.2 A logical handling of context in grammars

2.2.1 The need for a logical handling of context

As we have said before, Logic Programming has been involved with syntactic analysis since its very beginning. It was discovered that a formalism of context-free grammar rules augmented with uninterpreted first-order attributes and logical constraints is very powerful and can be straightforwardly translated into the formalism of Horn clauses programs (*DCG* for *Definite Clause Grammars* [12, 8, 48]). In this setting, the interpreter of Horn clause programs serves as a recognizing automaton.

The formalism of context-free grammars is in itself unable to express in detail the syntax of either programming or natural languages. It only offers us a fair compromise between precision in the expression and complexity of the parsing process. What is missing is *context*, but not necessarily the kind of context proposed by context-sensitive grammars, nor the kind of context proposed by concrete systems built around the formalism of context-free grammars. Context-sensitive grammars make the parsing process too complex, and concrete systems means are not grammatical. For instance, Yacc [30] and DCGs allows procedures and data structures in C, and in Prolog, respectively. DCGs' only grammatical handling of context is by means of terminal symbols to the right of the head non-terminal. This can be used to do some *look-ahead*, or to parse a phrase such as "aint" as "is not".

```
word, [" "] --> list_of_chars , [" "] .          % look-ahead
is(N), ["not"] --> ["aint"] .                    % aint = is not
```

So, handling context in DCGs is either non-logical or weak.

An example shows the need for a richer logical handling of the context. Pareschi and Miller's paper [44, 45, 47] showed that subtleties of the implementation of the *slash* feature of GPSGs which are only operationally explained in DCG implementation gain a neat logic-based explanation in intuitionistic logic. The idea of the slash feature is to observe that a relative clause is a relative pronoun followed by a declarative sentence with an elided component which is a noun-phrase. This is conventionally noted as

$$REL \rightarrow pronoun\ S/NP$$

One may either build up a specific set of rules for generating these incomplete declarative sentences, or share the description of what a declarative sentence is and describe the elided component. The slash feature invites us to do the second. This rule can be rephrased as "A list of words is a relative clause if, assuming the existence of an empty noun phrase, it can be parsed as a declarative sentence". This can be pictured by the following derivation rule.

$$\frac{Grammar, NP \rightarrow \epsilon \vdash S \rightarrow_* String}{Grammar \vdash S/NP \rightarrow_* String}\ slash$$

It is possible to derive *String* from *S/NP* if it is possible to derive the same *String* from S using the same grammar augmented with the new rule that an empty string can be derived from *NP*. This derivation rule can be compared with the right introduction rule for the implication connective in the sequent calculus.

$$\frac{Theory, A \vdash B}{Theory \vdash A \Rightarrow B}\ \Rightarrow_{\mathcal{R}}$$

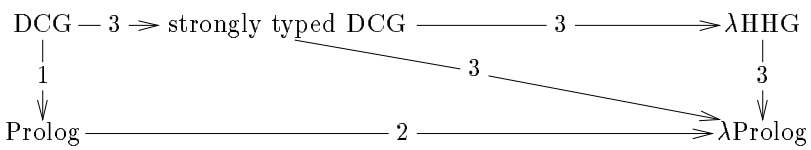
Thus, the derivability formula $S/NP \rightarrow_* String$ can be rendered by the logic formula $NP(\epsilon) \Rightarrow S(String)$. It happens that implication in right formulas (i.e. in goals) is one of the new connectives of λProlog. So, λProlog seems a good candidate for implementing the context handling implied by the slash feature.

The authors who first presented this technique of using implication for handling unbounded dependencies in GPSGs, Hodas, Miller, Pereira, Pareschi, warned of two problems. First, the assumption about the elided component must be effectively used once and only once in the parsing of the relative clause, but the logic of intuitionistic implication does not require it. An *ad-hoc* solution is to install a kind of hand-shaking protocol using terms and goals. This is in contradiction with our goal of handling context in a logical way. Another solution is to use the implication of Linear Logic [23, 26, 25, 27], which does enforce this constraint. In the latter solution the hand-shaking is done behind the scene.

Second problem, the elided component may not be used in the category implied by the pronoun that introduces the relative sentence. For instance, pronoun "whom" is an object case, but nothing forces the gap to be used as such. We believe it is due to a lack of precise categorization in these authors' examples. In a more full-fledged application, a set of agreement features is shared by the pronoun and the elided noun phrase; this ensures the elided component will be used in a category compatible with the pronoun. Controlling the proper usage of a contextual information using terms may look like a breach in our objective of handling context in a logical way. In fact, it is not; using agreement features is only a way of factoring out the common structure of several non-terminals.

2.2.2 Higher-order hereditary Harrop grammars

We call *higher-order hereditary Harrop grammars* (*λHHG*) the Logic Grammar formalism that imports from λProlog its clause structure. It makes λProlog into a Logic Grammar formalism. The relations between DCG and λHHG, and Prolog and λProlog can be illustrated by the following diagram:



1. Prolog [34], the DCG formalism [48], and the translation of DCG into Prolog by Prolog are classical[8, 43].
2. The evolution from Prolog to λProlog is due to Miller and Nadathur [37] and some advantages of using λProlog for Computational Linguistics are shown by the same authors [36], Pereira [47], and Pareschi and Miller [44, 45].
3. A formalism that is to DCG what λProlog is to Prolog is needed, as is the way to translate it into λProlog. λHHG is such a formalism. It is presented in another paper [29] with a bias towards formal languages. As an intermediary concern, we present a strongly typed DCG formalism and its translation into λProlog by λProlog.

Extending DCGs with λProlog features is useful because it combines the software technology advantages of DCGs and of λProlog: for DCGs, getting closer to a grammatical formalism, and automating the low-level operations of a syntactic analyser, and for λProlog, adding scope to terms, programs and resolution, while keeping in touch with logic. Another proposal for the merging of λProlog and Logic Grammar is Haas and Jayaraman's *higher-order DCG* [24]. The enrichment of higher-order DCG over plain DCG is that attributes are simply typed λ-terms. The rule language and the language for expressing conditions remain context-free rules and Prolog goals. There is no improvement in the way context is handled. Our proposal extends DCGs at the rule level for allowing a logical handling of context.

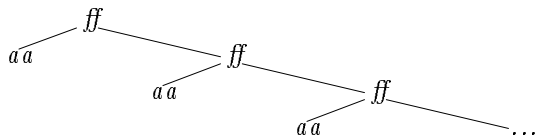
3 TWO ADVANCED LOGIC PROGRAMMING LANGUAGES

In this section we introduce the main features of Prolog II and λProlog. We will consistently use the concrete syntax of these two languages. It makes every example directly usable as a programming exercise, which we think overcomes the added difficulty of reading two different syntaxes. We assume a basic knowledge of the syntax and semantics of Prolog [8, 34].

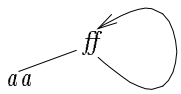
3.1 Prolog II

3.1.1 Semantics

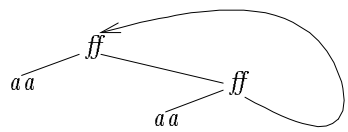
Initially, Prolog was a theorem prover based on Robinson's resolution principle [50]. In 1976, Kowalski and Van Emden [32] gave a theoretical model for Prolog: the Horn clauses. However, they proposed a unification algorithm involving the *occurrence-check*. As a variable cannot be unified with a term in which it occurs, this algorithm must check occurrences of variables. For efficiency reasons, most Prolog implementations do unification without the occurrence-check. The lack of an occurrence-check makes it possible to unify variable x and the term *ff(aa,x)*, resulting in the following infinite term:



This infinite term is called *rational* because it has only a finite number of different subterms (viz. itself and aa). So, it can be represented by finite graphs whose nodes are the subterms. There may be several such graphs representing the same rational term. The *minimal representation* of a rational term is the finite graph that has one node for every subterm.



Minimal representation



Non-minimal representation

Of course, this is not in accordance with the semantics of Prolog and this led Colmerauer to give new formal underpinnings for Prolog, turning a bug into a feature. In Prolog II and Prolog III, the notion of unification was replaced by the notion of resolution of *equations* and *inequations* on *finite and infinite rational trees* [13, 14, 10].

Rational trees have a characteristic both finite and infinite: they have a possibly infinite number of subtrees, but only a finite number of different subtrees. Their infinite characteristic is exploited in [16] for representing rational languages. Conversely, in the present work, we take advantage of their finite characteristic. We will always consider the minimal representation of a rational tree.

Prolog II also offers a built-in predicate that can be thought of independently from the rational trees, but that is pragmatically related to them. This predicate is the *dif* predicate, which expresses that two terms are not unifiable. It is an important means for exploring a rational tree without entering a loop.

3.1.2 Syntax

The syntax of Prolog II is roughly isomorphic to that of Prolog, though lexically very different. Identifiers are letters followed optionally by more letters and non-letters (digits, hyphens, or single quotes). Identifiers of variables are single letters, or have a non-letter in the second position. For instance, *a*, *a'*, *a1*, *a-b-c A*, *A'*, *A1*, *A-B-C* identify variables. Identifiers that have a letter in the second position identify constants.

The clause constructor is –>. It takes an atomic formula to its left, and a (possibly empty) sequence of atomic formulas terminated by a *';'* to its right. Constant *nil* is the empty list, and constant *'.'* is the binary list constructor. It has an infix operator syntax. Thus, the familiar "naive reverse program" is written as follows.

```
append(nil, l, l) -> ;
append(a.x, y ,a.z) -> append(x, y, z) ;
nrev(nil, nil) -> ;
nrev(a.l, r_la) -> nrev(l, r_l) append(r_l, a.nil, r_la) ;
```

In Prolog II, there is no special input syntax for rational trees. They are usually constructed using a system of equations. For instance, the following system constructs (the minimal representation of) the term described above.

```
eq(x, x) -> ;
sample-term(t) -> eq(t, ff(aa, t)) ;
```

3.2 λProlog

Miller proposes an extension of both the term language and the formula language of Prolog that still has nice proof-theoretic properties [37, 39, 38, 5].

3.2.1 Syntax

First, the extended term language is the language of the simply typed λ-terms. λ-Terms can be considered as functions, and simple types describe from what and to what domain they are defined. Simple types are generated by the following grammar[5]:

$$\mathrm{T} \quad ::= \mathcal{U} \;\mid\; \text{`('}\,\mathcal{K}_i\,\mathrm{T}^i\,\text{`)'} \;\mid\; \text{`('}\,\mathrm{T}\text{->}\,\mathrm{T}\,\text{`)'}$$

where $\mathcal{U}$ (resp. $\mathcal{K}_i$) are identifiers of type variables (resp. of type constructors of arity i). There is always a type constant o in $\mathcal{K}_0$ for the type of logical formulas. The arrow -> is supposed right-associative and unnecessary brackets can be dropped: e.g. *o->o->o* stands for *(o->(o->o))*.

Variables in types support the very important notion of *polymorphic typing*. Types with variables are in fact *type schemes*. The typing discipline is roughly similar to ML's [40] or to Typed Prolog's [42, 33]. In short, every constant has a type scheme, and all its occurrences must have a type which is an instance of the type scheme. Instances at different occurrences are independent.

Simply typed λ-terms are generated by the following grammar:

$$\Lambda_t \quad ::= \mathcal{C}_t \;\mid\; \mathcal{V}_t \;\mid\; \text{`('}\,\Lambda_{t'\to t}\,\Lambda_{t'}\,\text{`)'} \qquad t,\, t' \in \mathrm{T}$$
$$\Lambda_{t'\to t} \quad ::= \mathcal{V}_{t'} \setminus \Lambda_t \qquad t,\, t' \in \mathrm{T}$$

where $\mathcal{C}_t$ (resp. $\mathcal{V}_t$) are identifiers of constants (resp. of λ-variables) of type t. Note the type constraints in the attribute of the terminals and non-terminals. In the concrete syntax of λProlog, abstraction is denoted by an infix '\' rather than by a prefix 'λ': e.g. the identity function is written $x\backslash x$ instead of $\lambda x(x)$. Application is supposed left-associative and unnecessary brackets are dropped: e.g. *(append A B AB)* stands for *(((append A) B) AB)*.

Second, the syntax of programs ($\mathcal{P}$-formulas), definite clauses ($\mathcal{D}$-formulas) and goals ($\mathcal{G}$-formulas) is as follows:

$$\begin{array}{lll}
\mathcal{P} & ::= & \mathcal{D}.\,\mathcal{P} \;\mid\; \mathcal{D}. \\
\mathcal{D} & ::= & \mathcal{A} \;\mid\; \mathcal{A}\text{:-}\,\mathcal{G} \;\mid\; \mathcal{D},\,\mathcal{D} \;\mid\; pi\;\mathcal{V}_t \setminus \mathcal{D} \\
\mathcal{G} & ::= & \mathcal{A} \;\mid\; \mathcal{G},\,\mathcal{G} \;\mid\; \mathcal{G};\,\mathcal{G} \;\mid\; \mathcal{D}\text{=>}\,\mathcal{G} \;\mid\; pi\;\mathcal{V}_t \setminus \mathcal{G} \;\mid\; sigma\;\mathcal{V}_t \setminus \mathcal{G} \\
\mathcal{A} & ::= & \Lambda_o
\end{array}$$

The novelty is all in $\mathcal{G}$: explicit quantifications (both universal and existential, written *pi* and *sigma*) and implications (=>) are allowed in goals. These formulas are called *hereditary Harrop formulas*. Note that they form a strict superset of the *Horn formulas*.

As in Prolog, variables that are free in program clauses are considered universally bound at the clause level. They stand for unknown terms, and are called either *logical variables*, *unknowns*, or even *Prolog variables*. The lexical conventions and the syntax of connectives are the same as for Prolog as far as the first-order Horn formula fragment of λProlog is concerned. However, terms (even first-order) obey a curried syntax. Thus, the familiar "naive reverse program", with its type declaration, looks like the following.

```
kind list type -> type .            % Type constructor list of arity 1
type '[]' (list T) .                % T stands for an unknown type
type '.' T -> (list T) -> (list T) .
      % A.B and A.B.C are usually written [A|B] and [A,B|C]
      % A.[] and A.B.[] are usually written [A] and [A,B]
type append (list T) -> (list T) -> (list T) -> o .
type nrev (list T) -> (list T) -> o .

append [] L L .
append [A|X] Y [A|Z] :- append X Y Z .

nrev [] [] .
nrev [A|L] RLA :- nrev L RL , append RL [A] RLA .
```

[5] The language for expressing grammars uses round brackets at the metalevel. When a round bracket is used at the object-level it is single-quoted, e.g. '('.

Note the declarations. In λProlog, every constant in the sets $\mathcal{C}_t$ and $\mathcal{K}_i$ must be declared. The "kind" declaration introduces type constructors. The arity is denoted by the number i of arrows in the *(type->)^i type* expression: e.g. *list* has arity 1. The "type" declaration introduces constants and their type schemes. For instance, the type scheme of *'.'* is *T->(list T)->(list T)*, which means that *'.'* takes one argument of any type *T*, plus another that must be of type *(list T)*, and returns a result of type *(list T)*. So lists are polymorphic, but all the elements of a given list have the same type; they are called ***homogeneous polymorphic lists***.

The Prolog lexical convention helps categorizing free occurrences of identifiers: capitalized free occurrences stand for logical variables, others free occurrences stand for constants. Bound occurrences, either capitalized or not, stand for λ-variables.

3.2.2 Semantics

The terms of λProlog are simply typed λ-terms with prenex variables in types. This means that they obey the theory of simple types as presented by Church [7], augmented with a generic polymorphism capability as introduced by Milner [40] (see ML for another example of generic polymorphism).

The semantics of λProlog is usually based on proof theory [39, 38] rather than on model theory as for Prolog [34]. The main result is that a certain kind of goal-directed proofs, called *uniform proofs*, is complete with respect to intuitionistic provability for these formulas. In other words, every time a hereditary Harrop $\mathcal{G}$-formula is a consequence of a hereditary Harrop $\mathcal{D}$-formula, this can be proven using a uniform proof. In still other words, to be uniform is a restriction that rules out some proofs but that does not rule out any logical consequences among hereditary Harrop formulas.

Proof-theory expresses the operational reading of the new connectives as follows:

sigma — To prove a goal *(sigma v\G)*, select a new logical variable *V* that has the type of *v*, and prove the goal *G[v←V]*.

pi — To prove a goal *(pi v\G)*, select a new constant *c* that has the type of *v*, and prove the goal *G[v←c]*, taking care *c* is not captured by older unknowns.

=> — To prove a goal *(C => G)*, prove goal *G* in the program augmented by clause *C*. Clause *C* is added at the beginning of the program. This is important because the λProlog interpreter, as in Prolog, tries the clauses from the top to the bottom of the program. Clause *C* remains in the program only for the proof of *G*. It is suppressed when the proof is completed.

A single word describes the new capabilities of λProlog: *scope*. Abstraction brings scope to λ-variables in terms, explicit quantification brings scope to variables in formulas, and the deduction rules for universal quantification and implication bring scope to constants and clauses, respectively, in the execution process. Programming in λProlog is often a game of exchanging the representation of scope over the three levels. For instance, the structure of predicate *reduce* (see section 4.2) is based on the correspondence between the three kinds of scopes.

3.2.3 The manipulation of λ-terms

The theories of λProlog and higher-order unification make it impossible for a logical variable or a predicate argument to become bound to some term with a free λ-variable in it[6]. In other words, logical variables and predicate arguments can only be bound to closed λ-terms (*combinators*). So, the only way to access the body of an abstraction *A* without capturing free occurrences of the abstracted λ-variable is to apply the abstraction to some term *t*, that which will "consume" the λ-abstraction.

For this operation to be correct, it must not map non-λ-equivalent terms to λ-equivalent terms. This is achieved through two conditions at the metalevel: first, term equivalence must include η-equivalence[7], second, the term *t* must be a universal variable that is quantified in the scope of abstraction *A*. This corresponds to the fact that an abstraction *λx(E)* can be interpreted as the function that maps in a unifirm way *every* term *t* onto the term *E[x←t]*. So, abstraction can be viewed as a kind of universal quantification in terms, and universal quantification in $\mathcal{G}$-formulas is its necessary counter-part. So, one can reconstruct an abstraction by solving

[6] In that respect, logical variables differ from the syntactic variables used in texts about the λ-calculus like the present article.

[7] We recall the η-equivalence axiom: $\lambda x(E\ x) =_\eta E$, if x does not occur free in E. The practical consequence of this axiom is that two functions, with different definitions, that always return equivalent outputs when given equivalent inputs, are considered equivalent also. This equivalence cannot be proven using αβ-equivalence alone. To η-reduce is to use the axiom as a rewriting rule from left to right.

problems of the following form: $\exists A\ \exists X\ \forall t\ (A\ t)=(X\ t)$. The solution in X of the problem is the reconstructed abstraction.

In λProlog like in Prolog, the theory of the constants is defined by the program. However, universal constants require a special treatment because they are scoped terms. One must add to the program the theory of every universal constant only in the scope and for the lifetime of the constant. This is done using implication in goals. Predicate *reduce* (see section 4.2) shows a concrete example of the conjoint use of universal quantification and implication for traversing abstractions.

The remark that logical variables can only be bound to combinators makes every occurrence in a program of a term like $x\backslash E$ suspect. In λProlog, an abstraction with no explicit occurrence of its bound variable in its body is suspect in the same way as in Prolog an unknown with only one occurrence is suspect: one needs not even know the application. Such an abstraction is not illegal. It represents exactly all functions that ignore their argument (i.e. *constant functions*), but these functions are seldom of interest. However, they can be found in print where constant functions are not intended (e.g. [51], pp. 263, 265–269). All this shows that programming with λProlog requires new reflexes that are not so well known.

3.2.4 Context in programs

We give as an example of context in programs the λProlog code of an unfamiliar implementation of the familiar concatenation relation.

```
type app (list T) -> (list T) -> o .
type append (list T) -> (list T) -> (list T) -> o .

app [A|X] [A|Z] :- app X Z .
append X Y Z :- (app [] Y => app X Z) .

% Usual definition:
% append [] X X .
% append [A|X] Y [A|Z] :- append X Y Z .
```

In the usual definition, the second argument in predicate *append* is transmitted untouched through parameter Y until it is used by the first clause. We may say that Y conveys the second argument of the relation to the terminal case.

In the unusual definition, the second argument is conveyed to the terminal case via the asserted clause *(app [] Y)*. To convey context through the program than through terms is often more concise because clauses have names and parameters have only positions so that accessing context involves a naming instead of an explicit search. It also allows us to mention the context only when needed because the program is always an implicit context. Moreover, it raises the context to the formula domain, allowing reasoning based on the deduction rules of intuitionistic logic.

4 REPRESENTING HIGHER-ORDER TERMS

As we have seen in section 2.1, a solution to deal logically with the compositional construction of semantic formulas, and with the quantifiers they may contain is to take abstraction and λ-calculus seriously.

The automatic manipulation of λ-expressions involves a great deal of work, both in machine time and in programming effort, because some bound variables must be renamed in order to avoid *captures* when doing substitutions. As far as λ-expressions are concerned, every bound variable (i.e. in the scope of λ) can be renamed. For instance, $\lambda x\lambda y(y\ x\ \lambda y(x\ y))$ and $\lambda u\lambda v(v\ u\ \lambda w(u\ w))$ are syntactically different expressions that denote the same function. They are called *α-equivalent*[8] and can be identified [4].

The need for renaming arises in β-reductions because every occurrence of a variable in one expression has to be replaced by another expression. Some free variable of the second expression might have the same name as a bound variable of the first one, with the effect (called a *capture*) that a binding is introduced where it is not intended. For instance, a careless β-reduction of $\lambda x(\underline{\lambda y}\lambda x(\underline{y})\ \underline{x})$ yields $\lambda x\lambda x(x)$ which is the function that takes two parameters and returns the second one, instead of $\lambda x\lambda z(x)$ which is the function that takes two parameters and returns the first one. So, one must rename bound variables of the left part of a redex.

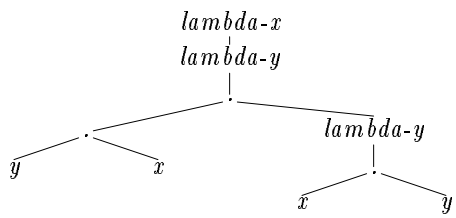
[8] We recall the α-equivalence axiom: $\lambda x(E) =_\alpha \lambda y(E)[x \leftarrow y]$, if y does not occur bound in E. The intuition of this axiom is the consistent renaming of formal parameters.

We present here three methods for implementing λ-terms. The first one handles the problem with first-order terms, but taking advantage of the rational trees of Prolog II and Prolog III. The last two methods propose to leave the first-order world and to use the λ-expressions of λProlog.

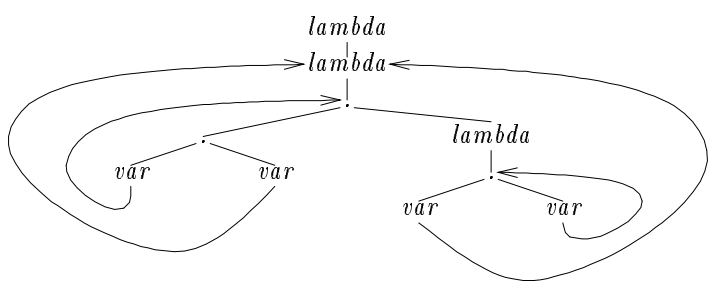
For every method, we give the implementation of the calculus by a normalizer. It is always very concise, and reasonably efficient. It must be mentioned however, that it may not terminate for non-strongly normalizable expressions (i.e. expressions that can start an infinite reduction). A way to prove strong normalization is to prove that the λ-terms we deal with can be simply-typed.

4.1 Object-level λ-expressions as rational trees

We can bypass the problem with bound variables names by getting rid of them and by working on α-equivalence classes using an adequate notation. De Bruijn has proposed a representation for these α-classes [20]. We present here another coding relying on rational trees, which makes an automatic treatment of λ-expressions in Prolog II easy. A λ-expression can be classically represented by a finite first-order term whose leaves are variables or constants and whose internal nodes are labelled by unary symbols of the form *lambda-x* (x being a variable name) or by the binary symbol, '.', denoting the *application* of an expression to another one. Thus, expression $\lambda x \lambda y(y\ x\ \lambda y(x\ y))$ can be viewed as the finite tree



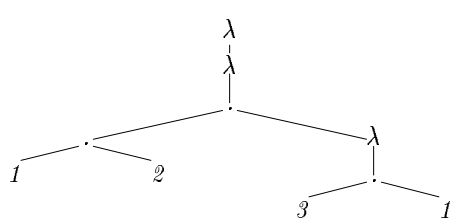
Informally, our representation amounts to transforming such a tree into an infinite one. Every bound variable leaf is replaced by a node labeled by the unary symbol *var* (meaning that it denotes a variable), whose only son is the tree that represents the scope of its binding λ. Therefore, it is no longer necessary to recall the name x in the symbol *lambda-x*. Only one unary symbol *lambda* is required. The above expression is coded by the following rational tree:



It is the solution in t of the following system of equations (a Prolog II goal):

t=lambda(x) x=lambda(y) y=(var(y).var(x)).lambda(z) z=var(x).var(z)

This representation can be compared to graph-based representations in which positions stand for names. It can be also compared with that of De Bruijn. In De Bruijn's system, the term is represented by the term below, in which every occurrence of an integer i denotes a variable bound by the ith λ on the path from this occurrence to the root.



Our representation is more suitable for Logic Programming because the renaming problem raised by β-reductions is not handled by counting the nodes labelled *lambda*, as it is the case with De Bruijn's notation (see [53] for general algorithms for this notation). The renaming problem is solved by the unification of rational terms, which is automatically performed by Prolog II. Thus, the resulting normalization algorithm[9] is particularly concise. The missing definitions like *member* already belong to any Logic Programming library.

```
normalization(e,e') -> reduce(<e,e">,nil) start-again(e,e",e') ;                    % 1
start-again(e,e,e) -> ;
start-again(e,e",e') -> dif(e,e") normalization(e",e') ;

% reduce(<t,s> , <var(m0),n0>.<var(m1),n1>. .... . nil)
% t reduces to s, knowing that the ni's are substituted to the var(mi)'s.
reduce(<var(m),a>,s) -> member(<var(m),a>,s) ;
reduce(<c,c>,s) -> constant(c) ;
reduce(<lambda(m),lambda(m')>,s) -> reduce(<m,m'>,<var(m),var(m')>.s) ;             % 2
reduce(<lambda(m).a,m'>,s) -> reduce(<a,a'>,s) reduce(<m,m'>,<var(m),a'>.s) ;       % 3
reduce(<f.a,f'.a'>,s) -> dif-from-lambda(f) reduce(<f,f'>,s) reduce(<a,a'>,s) ;

dif-from-lambda(var(m)) -> ;
dif-from-lambda(f.a) -> ;
dif-from-lambda(c) -> constant(c) ;
```

The normalization process is performed in several steps. Each step corresponds to a call to predicate *reduce* in clause 1. It consists in traversing expression *e* and β-reducing each redex in it, producing *e'*. Now, this operation can create new redexes. Thus, it must be iterated (predicate *start-again*) until the resulting expression is in normal form. The second argument in predicate *reduce* is a list of pairs *<var(m),n>* denoting the substitution of *n* to the variable *var(m)*. This list is used both to perform β-reduction (clause 3 in predicate *reduce*) and to prevent from looping while traversing the infinite tree *e* (clause 2 in predicate *reduce*). Remember that traversing a *lambda* amounts to step into a loop. For further details on this algorithm, its proof, and the soundness of the coding, see [15].

4.2 Object-level λ-expressions as λProlog's λ-expressions

If the object-level terms are simply typed λ-terms, and we do not want to exert any specific control on them, we can directly use the λProlog notation. Thus, both normalization and substitution of a term for a λ-variable are handled transparently by λProlog. The usual prefix 'λ' is written as an infix '\', so that *λx(x)* is written *x\x* and *λxλy(y x λy(x y))* is written *x\(y\((y x) (y\(x y))))*. Most parenthesis being redundant, *x\y\(y x y\(x y))* denotes the same term. We call this the *direct* implementation in λProlog.

One may wish to use a more explicit representation in which applications and abstractions are tagged, but substitution is performed transparently. It is a middle course between the λ-term-as-rational-tree representation and the direct notation in λProlog. One reason for preferring a more explicit representation is the need to exert a control on the λ-terms that is different from λProlog's. For instance, the typing may be more sophisticated than simple typing, and allow for inheritance [3], while higher-order terms are still needed for handling compositionality and quantifiers.

With this explicit representation two dedicated constructors (say '.' and *lambda*, to stress the resemblance with the λ-term-as-rational-tree representation) are used for labelling applications and abstractions. They are declared as follows:

```
kind lt type .
type '.' lt -> lt -> lt .
type lambda (lt->lt) -> lt .
```

Unlike the λ-term-as-rational-tree representation, λ-variables are not tagged. Thus, the expression *λxλy(y x λy(x y))* is represented by the λProlog term *(lambda x\(lambda y\((y.x).(lambda y\(x.y)))))*.

We call this the *explicit* representation in λProlog. It cannot be transparently normalized, or even β-reduced, by λProlog. For instance, λProlog cannot rewrite *((lambda x\x).F)* into *F* since the *lambda* blocks the reduction.

[9] This one is a call-by-value normalizer. Call-by-name is concise as well.

However, programming the normalizer is even easier than with the λ-term-as-rational-tree representation. The reason is that substitutions of a term for a variable is transparently handled by λProlog's β-reduction.

Follows the code of a normalizer for the explicit representation in which we consistently use the notation of the Prolog II normalizer wherever possible. Note the use of a universal quantification and implication at the goal level for interpreting abstractions in clause 1 of *reduce*, and the use of λProlog's β-reduction in term *(M A_1)* in clause 2. Using λProlog's β-reduction eliminates the need to manage a substitution list in predicate *reduce*.

```
type (normalization, reduce) lt -> lt -> o .
type start_again lt -> lt -> lt -> o .

normalization E E_1 :- reduce E E_2 , start_again E E_2 E_1 .
start_again E E E .
start_again E E_2 E_1 :- dif E E_2 , normalization E_2 E_1 .

reduce C C :- constant C .
reduce (lambda M) (lambda M_1) :- pi x\(constant x => reduce (M x) (M_1 x)) .        % 1
reduce (lambda M).A M_1 :- reduce A A_1 , reduce (M A_1) M_1 .                        % 2
reduce F.A F_1.A_1 :- dif_from_lambda F , reduce F F_1 , reduce A A_1 .
```

This program is an example of a general λProlog programming scheme for representing object-level scoped data-structures. The scheme is fourfold: first, to use λProlog's abstraction to represent scope and substitutability of object-level variables, second, to use λProlog's application to implement substitution, third, to use universal quantification to access abstractions bodies, and fourth, to use implication to qualify universal variables. The first two parts of the scheme are just natural; λProlog's abstraction acts as a generic quantification, and β-reduction acts as a substitution operation that handles scope properly. The third and fourth parts require more attention. They are explained in section 3.2.3.

The explicit representation applies the scheme as follows. Object-level abstractions are represented by λProlog's abstractions because they introduce scope and substitutability. This is the first part of the scheme. Clause *2* of predicate *reduce* uses λProlog's application for implementing object-level substitution. This comes from the second part of the scheme. Clause *1* of predicate *reduce* propagates reduction through abstractions using universal quantification and implication. It implements the third and fourth parts of the scheme.

4.3 Comparison of the proposed representations

When presenting the three different representations for object-level λ-terms we have focused on the normalization process. We now compare them with respect to typing and unification.

4.3.1 Typing

Typing is reflected quite differently in the various implementations. The Prolog II implementation has no concern for types of any kind. The explicit λProlog implementation actually maps object-level λ-terms onto untyped λ-terms[10]. So, though λProlog is strongly typed, there is no concern for the types of object-level terms either. Since the direct λProlog implementation maps the object-level terms onto their λProlog notation, types are preserved and checked at compile time. This is the only version that can detect a typing error in the manipulation of object-level terms without supplementary programming. The other versions need a type checker.

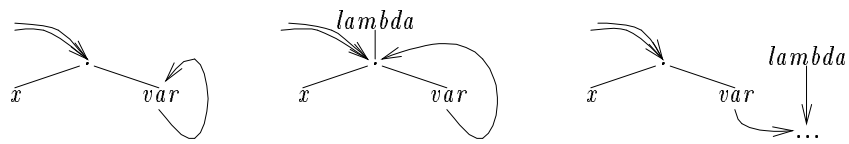
As for the representation of terms, a type checker may be *direct* of *explicit*. If the object-level types are simple types, the direct representation is to incorporate them in the term constructors of the explicit representation of terms as follows:

```
kind lt type -> type .
type '.' (lt A->B) -> (lt A) -> (lt B) .
type lambda ((lt A)->(lt B)) -> (lt A->B) .
```

So doing, the object-level type checker is the metalevel type checker. However, if the object-level types are not simple types, or if one must exert some control on type-checking, one must represent object-level types explicitly as metalevel terms.

[10] More precisely, all terms of the representation have the same type *lt*.

The λ-term-as-rational-tree representation is more permissive than the explicit representation: one can build with constants '.', *var*, and *lambda* rational terms that represent no object-level terms. For instance, the terms pointed to by double arrows in



are perfectly well formed rational terms, but they do not represent any well formed object-level term. The leftmost one is not even a part of a well formed representation because the subterm of a *var* can only be a subterm of some *lambda*. Note that well-formedness depends on one's point of view. For instance, the central term is a well formed representation as a whole, but the part pointed to by the double arrow is not a well formed representation. Even if constants '.', *var*, and *lambda* are typed, this cannot prevent us from building badly formed representations.

4.3.2 Unification

The most visible difference between the three representations is related to unification. According to the representation, object-level terms are unified modulo either the theory of rational terms, the theory of simply typed λ-terms with $\alpha\beta\eta$-equivalence, or the theory of untyped λ-terms with α-equivalence.

What unification can be used for also depends on the representation. The λ-term-as-rational-tree representation and the explicit representation allow us to check the syntactic structure of terms by unifying them with suitable patterns. A set of structure-checking predicates can be written either in Prolog II or in λProlog.

```
is-application( a.b ) -> ;                          % Prolog II
is-abstraction( lambda(t) ) -> ;
is-variable( var(t) ) -> ;

is_application (_ . _) .                            % λProlog: explicit representation
is_abstraction (lambda _) .
```

The implementation of *is_variable* for the explicit representation in λProlog is less direct because logical variables cannot capture λ-variables. Relation *is_variable* must be defined for every traversed abstraction. For instance, the following predicate updates the definition of *is_variable* every time it traverses an abstraction.

```
p (lambda E) :- pi x \( is_variable x => p (E x) ) .
```

A constructor *var* for object-level variables would make this test easier, but it would bar the transparent substitutability offered by λProlog.

The direct representation in λProlog does not allow us to check the structure of a term simply because terms with different structures may be $\alpha\beta\eta$-equivalent.

What form solution substitutions of unification problems can take also depends on the representation. As a general rule of λProlog, binding values can only be combinators. This affects differently the two λProlog representations. The direct representation simply transposes the restriction at the object level. The explicit representation makes it possible to bind E to *x\(f x)* when unifying *(lambda E)* and *(lambda x\(f x))*. At the object level the term *x\(f x)* can be considered having a free variable, x, that can be named using universal quantification. See, for example, the predicate p defined above. The λ-term-as-rational-tree representation imposes almost no restriction. For instance, the three invalid representations pictured above can all be built as solutions of some unification problem.

4.3.3 How to choose a representation?

As we have seen above the three proposed representations for λ-terms are not equivalent.

The direct representation seems by far the most convenient because it offers built-in the type-checking, normalization, and unification of object-level λ-terms. However, the object-level λ-terms must be similar to the metalevel terms in every respect, and the application must not require to exert any control of any kind.

As soon as either the application requires a specific control on object-level terms (to specify a reduction strategy, to decide whether an object-level term is an abstraction or not, etc), or the object-level λ-terms are not exactly similar to the metalevel terms, one must revert to the explicit representation. Using the explicit representation of terms, one has to choose between the direct and explicit representation of types. The criteria are the same as for terms. If object-level types are similar to metalevel types in every detail, then the direct representation works, otherwise, one must use the explicit representation.

The place of the λ-term-as-rational-tree representation is different. Its usefulness does not come from the failure of another representation. Its first interest is theoretical: It offers a first-order Logic Programming alternative to De Bruijn's nameless representation without the need for complex computation of indexes. Its second interest is that rational terms are a very particular class of feature terms (e.g. ψ-terms [3]). Feature terms generally offer partially ordered sort constructors, attributed labels, and coreference constraints, but rational terms offer an equivalent to coreference constraints only. So, the λ-term-as-rational-tree representation can be easily adapted to represent λ-terms with feature terms. It happens that feature terms are also useful for Computational Linguistics: for instance, for implementing Unification Grammars [31, 51]. So, the λ-term-as-rational-tree representation adapted to feature terms allows us to handle several facets of a Natural Language application in a unique framework. An alternative would be to provide λProlog with feature terms; this does not exist yet.

5 A LOGICAL HANDLING OF CONTEXT

DCGs can deal with any kind of context-sensitive construct using just Prolog terms and goals. However, it is advantageous to have λProlog's context management capabilities available at the rule level.

5.1 DCG

Given some grammar, the problem of syntax analysis is to build an automaton for recognizing that some input belongs to the language generated by the grammar. One wants to do better than using the grammar to generate words and then checking them with the input. A better way of controlling the generation is to make the input and the generated words share the same data-structure.

Assuming that the input to be analysed is a Prolog list of tokens[11], the clause that recognizes an interrogative sentence *In* made of a subject phrase *S* followed by an interrogative verb phrase *V* sharing agreement features *A* (as generated by the grammar of appendix A) can easily be written in Prolog.

```
int_sentence(In) :- append(S, V, In) , subj_n_phrase(A, S) , verb_phrase(int, A, V) .
```

Using *difference lists* [54], one no longer needs to split *In* with a call to *append*.

```
int_sentence(In, Out) :- subj_n_phrase(A, In, L) , verb_phrase(int, A, L, Out) .
```

The general structure of the grammar rule for interrogative sentences is straightforwardly reproduced in the clause. However, house-keeping makes the Prolog expression less readable than the grammar rule. The relative positions of the input items are made explicit in the Prolog clause by input/output parameters: the *linking variables* (here, variable *L*). The grammar rule specifies the positions of the input items implicitly by the relative positions of the terminals and non-terminals.

The goals of a Logic Grammar formalism such as DCGs are to hide the house-keeping, to keep close to a well-known grammatical formalism (context-free grammar), to enable contextual control, but still to be readily translatable into a Logic Programming language. The usual translation is very direct, and the completeness of the parsing scheme depends on the strategy of the Logic Programming interpreter. The usual SLD strategy is not complete, but a tabled strategy like the OLDT strategy yields a complete parser under some conditions [55].

The syntax of DCGs can be defined as follows:

$$
\begin{array}{lll}
\mathcal{G} & ::= & \mathcal{R}\ .\ \mathcal{G} \quad | \quad \mathcal{R}\ . \\
\mathcal{R} & ::= & \mathcal{H}\ \text{-->}\ \mathcal{B} \\
\mathcal{B} & ::= & \mathcal{NT} \quad | \quad \mathcal{T} \quad | \quad ! \quad | \quad \{\ \mathcal{FOG}\} \quad | \quad \mathcal{B}\ ,\ \mathcal{B} \quad | \quad \mathcal{B}\ ;\ \mathcal{B} \\
\mathcal{H} & ::= & \mathcal{NT} \quad | \quad \mathcal{NT}\ ,\ \mathcal{T} \\
\mathcal{NT} & ::= & \mathcal{C}_i\ \text{`('}\ \mathcal{FOT}\ (\ ,\ \mathcal{FOT}\ )^{i-1}\ \text{`)'} \\
\mathcal{T} & ::= & [\ ((\mathcal{FOT}\ ,\ )^*\ \mathcal{FOT}\ )^{0|1}\ ]
\end{array}
$$

[11] This need not be so, but we will assume it throughout section 5.

where $\mathcal{G}$ (resp. $\mathcal{R}$, $\mathcal{B}$, $\mathcal{H}$, $\mathcal{NT}$, and $\mathcal{T}$) stands for grammars (resp. rules, rule bodies, rule heads, non-terminal and terminal items). Connectives -->, ',' and ';' stand for derivation, concatenation and alternation. First-order terms in non-terminals, $\mathcal{FOT}$, play the rôle of *attributes*, and first-order goals in rules bodies, $\mathcal{FOG}$, play the rôle of *side-conditions*. We call them *condition goals*.

The rule for generating interrogative sentences made of a subject phrase and an interrogative verb phrase can thus be written in DCGs as:

int_sentence --> subj_n_phrase(A) , verb_phrase(int, A) .

When used in an analyser, this rule checks the syntactic correctness of an input string, but it returns no information on its content. A more useful variant of the rule builds a semantic formula. Whether the semantic formula is represented as rational trees (section 4.1), or simply typed λ-terms (section 4.2) is hidden in the details of predicate *application*.

int_sentence(S) --> subj_n_phrase(A, SNP) , verb_phrase(int, A, VP) , { application(SNP, VP, S) } .

Most Prolog implementations contains a preprocessor for translating DCG rules into Prolog. The rule above translates into the following clause:

```
int_sentence(S, In, Out) :-
        subj_n_phrase(A, SNP, In, L1) , verb_phrase(int, A, VP, L1, L2) ,
        application(SNP, VP, S) , L2 = Out .
```

The implementation of DCGs in an actual Prolog system follows two architectural rules.

1. The syntax of DCG rules is plain Prolog syntax: DCG rules are plain Prolog terms endowed with a specialized interpretation. So, DCG rules are first read in Prolog, then interpreted (most often compiled). The Prolog system recognizes these terms by their main connective ('-->' for DCG) and translates them according to rules that are written in Prolog.

2. DCGs are only one among the many formalisms that can be preprocessed in this way. There is a general "hook" for implementing such formalisms.

5.2 The transposition of DCGs into λProlog

In this section, we use λProlog instead of Prolog for implementing DCGs. The important point is that we want to stick to the architectural rules we mentioned above. We observe that the DCG syntax is not readable in λProlog, but that a variant of this syntax is readable.

5.2.1 A strongly typed variant of DCGs

The reason why the DCG syntax is not readable in λProlog is that DCG rules considered as terms are definitely ill-typed [5]. For instance, the concatenation connective, ',', is used to connect terminals (encoded as lists), goals (encoded as Prolog goals), and non-terminals. Since λProlog is a strongly typed language, the binary connective ',' must have a type *T->T->T* such that *T* is equal to the types of goals (i.e. *o*), lists (i.e. *(list E)* for some *E*), and non-terminals. This constraint alone cannot be satisfied; that the binary connective ',' is already used in λProlog with type *o->o->o* is only an aggravating factor. So, the DCG syntax is ill-typed and a renaming of its connectives is still ill-typed.

One has to find a common type for terminals, non-terminals, and goals. The type we propose is based on the consideration of the meaning of terminals and non-terminals, and how they are translated into Prolog.

The meaning of a terminal or non-terminal is a word between two positions in an input string. The way a non-terminal is translated into a Prolog goal is by adding two arguments. Because a Prolog identifier may belong to different arities, a term constructor *non_terminal/i* can be translated into the predicate constructor *non_terminal/i+2*. But a λProlog identifier belongs to only one type. λProlog's λ-terms give a simpler yet logical means for translating non-terminals into goals: if a non-terminal is a function that given two positions returns a goal, then the translation is done by applying the non-terminal to two suitably chosen positions.

We now consider that non-terminals have type *(list T)->(list T)->o* [12]. This is the type of a binary relation on homogeneous lists. It can also be seen as the type of a goal with two arguments missing. So, it corresponds to

[12] The assumption that inputs are lists is important here. However, if inputs are not lists the above type can be replaced by something else that characterizes input segments.

both the meaning and the translation of non-terminal items. It remains to force terminals and condition goals into that type.

To accomplish this, goals are wrapped in the term constructor *epsilon* (for *empty* non-terminal), and lists of terminals are wrapped in the term constructor '`'. Syntactic items are built with the connectives '&' for concatenation and ':' for alternation. The types and operator declarations of these connectives are as follows:

```
type ` (list T) -> ((list T)->(list T)->o) .
type epsilon o -> ((list T)->(list T)->o) .
type ('&', ':') ((list T)->(list T)->o) ->((list T)->(list T)->o) -> ((list T)->(list T)->o) .
type '-->' ((list T)->(list T)->o) -> ((list T)->(list T)->o) -> o .
op 1200 xfx '-->' . op 1000 xfy '&' . op 1100 xfy ':' .
op 950 fy 'epsilon' . op 850 fy ` .
```

The syntax of the strongly typed variant of DCGs is as follows:

$$
\begin{array}{lll}
\mathcal{G}_T & ::= & \mathcal{R}_T \ . \ \mathcal{G}_T \ \ | \ \ \mathcal{R}_T \ . \\
\mathcal{R}_T & ::= & \mathcal{H}_T \ \text{-->} \ \mathcal{B}_T \\
\mathcal{B}_T & ::= & \mathcal{NT}_T \ \ | \ \ \mathcal{T}_T \ \ | \ \ \mathcal{B}_T \ \& \ \mathcal{B}_T \ \ | \ \ \mathcal{B}_T : \mathcal{B}_T \ \ | \ \ epsilon \ \mathcal{FOG} \\
\mathcal{H}_T & ::= & \mathcal{NT}_T \ \ | \ \ \mathcal{NT}_T \ , \ \mathcal{T}_T \\
\mathcal{NT}_T & ::= & (\mathcal{C}_{t_1 \text{->} \ldots \text{->} t_i \text{->} (list\ T) \text{->} (list\ T) \text{->} o} \ \mathcal{FOT}^i)_{(list\ T) \text{->} (list\ T) \text{->} o} \\
\mathcal{T}_T & ::= & ` \ \mathcal{FOT}_{(list\ T)}
\end{array}
$$

The attribute T specifies the type of the input tokens throughout the grammar.

Thus, the strongly typed version of the DCG rule for generating interrogative sentences made of a subject phrase and an interrogative verb phrase is:

```
int_sentence S --> subj_n_phrase A SNP & verb_phrase int A VP & epsilon (application SNP VP S) .
```

5.2.2 The translation of strongly typed DCGs into λProlog

We present at length the λProlog way of translating DCGs into λProlog because it handles object-level variables (variables in the source rules and in the target clauses) more logically than what the Prolog way does [8, 43], and it is of some value as soon as λProlog's capabilities are available, even for implementing a different Logic Grammar formalism.

The λProlog rendering of the DCG rule for generating interrogative sentences is:

```
int_sentence S In Out :-
      sigma L1\( subj_n_phrase A SNP In L1 ,
      sigma L2\( verb_phrase int A VP L1 L2 , application SNP VP S , L2 = Out )) .
```

The main difference between the λProlog translation and the Prolog translation is in the explicit quantification of the linking variables. They can be quantified either universally at the clause level, or existentially at the goal level. To quantify them at the clause level requires knowing their number from the start of the translation in order to produce the proper number of quantifications. The goal level quantification allows us to produce the quantifications one at a time, when needed.

The translation is performed by induction on the structure of DCG rules: rule, head, body, terminal. Each case is handled by a dedicated translation predicate typed as follows:

```
type dcg_rule o -> o -> o .
type (dcg_head, dcg_body) ((list T)->(list T)->o) -> ((list T)->(list T)->o) -> o .
type dcg_terminal (list T) -> ((list T)->(list T)->o) -> o .
type dcg_pushback (((list T)->(list T)->o) -> ((list T)->(list T)->o)) -> o .
```

The case "body" is itself split into subcases according to the connectives a given body contains. To each connective corresponds a structure in the target language. These structures can be seen as the instructions of an abstract machine for executing DCGs. Each structure is definable by a combinator as follows[13]:

[13] The Prolog/Mali implementation of λProlog allows macro definitions that are interpreted by a preprocessor. Combinators used for translating connectives are defined this way.

#define CONJ left\right\in\out\(sigma Link\(left in Link, right Link out))

Connects two items in a sequential conjunction through a local variable *Link*.

#define DISJ left\right\in\out\(left in out; right in out)

Connects two items in a parallel disjunction.

#define EPSILON goal\in\out\(goal, in=out)

Fills in a goal of a clause and specifies that the input is not changed.

Our purpose is to compile DCGs into λProlog. However, it may be that parts of a DCG rule are undefined at compile-time[14]. So, an interpreter of the source connective must remain in the target program for executing the parts of the DCG that are only known at run-time. The interpreter is defined using the translation combinators as follows:

```
(` Terminal0) In Out :- dcg_terminal Terminal0 Terminal , Terminal In Out .
(Body1 & Body2) In Out :- CONJ Body1 Body2 In Out .
(Body1 : Body2) In Out :- DISJ Body1 Body2 In Out .
(epsilon Goal) In Out :- EPSILON Goal In Out .
```

The translation combinators are also used in the translation proper to aggregate the translations of the parts and to build ultimately a λProlog clause.

To translate a rule amounts to stripping the rule of its universal quantifications, and then splitting the rule into its head and its body. A "pushback transformer" (a combinator) associated to terminals in the head is passed through the program context (via implication). The *dynamic* directive warns the compiler that predicate *dcg_pushback* is subject to implication.

```
dynamic dcg_pushback .
dcg_rule (pi Rule) (pi Clause) :- pi y\(dcg_rule (Rule y) (Clause y)) .
dcg_rule (Head0 --> Body0) (pi In\(pi Out\(Head In Out :- PushBack Body In Out))) :-
      (dcg_pushback PushBack => dcg_head Head0 Head) , dcg_body Body0 Body .
```

The target clause has all the variables of the source rule plus two: every *pi* around the rule translates to a *pi* around the clause, and two *pi*'s are added for the input/output parameters.

The translation of a rule head specifies a pushback transformer to be applied to the clause body. The transformer is either a "conjunctor" or the identity function.

```
dcg_head (Head0 & (` Terminal0)) Head0 :- ! ,
      dcg_terminal Terminal0 Terminal , dcg_pushback body\(CONJ body in\out\(Terminal out in)) .
dcg_head Head0 Head0 :- dcg_pushback x\x .
```

To translate a body is to apply a combinator according to the main connective and to proceed.

```
dcg_body (Left0 & Right0) (CONJ Left Right) :- ! , dcg_body Left0 Left , dcg_body Right0 Right .
dcg_body (Left0 : Right0) (DISJ Left Right) :- ! , dcg_body Left0 Left , dcg_body Right0 Right .
dcg_body (epsilon Goal0) (EPSILON Goal0) :- ! .
dcg_body (` Terminal0) Terminal :- ! , dcg_terminal Terminal0 Terminal .
dcg_body NonTerminal0 NonTerminal0 .
```

The translation of a terminal is the only part that has to do with lists. It can easily be changed to handle other kinds of input.

```
dcg_terminal Terminal0 in\out\(in=(FTerminal out)) :- list_flist Terminal0 FTerminal .
```

Predicate *list_flist* implements the relation between a Prolog list (e.g. *[1,2]* with type *(list int)*) and a functional list (e.g. *z\[1,2|z]* with type *((list int)->(list int))*).

[14] E.g. a DCG rule for emulating Kleene's star:

```
kleene_star Non_terminal --> ` [] : Non_terminal & kleene_star Non_terminal .
```

```
type list_flist (list T) -> ((list T)->(list T)) -> o .
list_flist L FL :- pi list\(append L list (FL list)) .
```

The predicate says that *FL* is the functional version of *L* if for every *list* the concatenation of *L* and *list* is the application of *FL* to *list*.

Checking the instantiation of grammatical items must be added to make the system robust. All the programs have been compiled and executed with the Prolog/Mali implementation of λProlog. Any other implementation will do except for such facilities as macro definitions of combinators.

5.3 Higher-order hereditary Harrop grammars

5.3.1 Discussion

Since our purpose in the previous sections was only to give a strongly typed variant of DCGs, we have used first-order goals in their definition, and we have not insisted on using higher-order terms.

In fact, what kind of terms and goals are used for attributes and conditions in a DCG is irrelevant to the translation. Attributes and conditions are merely plugged into the target clauses. If one uses simply typed λ-terms instead of first-order terms, then DCGs become Haas and Jayaraman's *higher-order DCGs* [24]. One may also use λProlog goals instead of Prolog goals as side-conditions.

Our experience, in the domain of formal languages as well as of natural languages (e.g. section 2.1), is that allowing λProlog terms and goals is a bonus. As another example, consider how left-recursive attributed rules are transformed into non-left-recursive rules. The resulting rules often have additional attributes for implementing an "accumulator", which is in fact a restricted form of function abstraction and composition (see also a non-Prolog version of this in [2], pp. 302–305).

However, allowing λProlog terms and goals is not enough to propagate the logical handling of context from λProlog to the grammatical level.

In Pareschi and Miller's example [45], a relative sentence *rel* contains a declarative sentence *s* in which a noun phrase *np* is elided (a *gap*). They propose to code rule *REL* → *whom S/NP* by the first of the two following λProlog clauses.

```
rel REL ["whom" |In] Out :- pi gap\( np gap Z Z => s (REL gap) In Out ) .

rel REL ["whom" |In] Out :- pi gap\( pi Z\(np gap Z Z) => s (REL gap) In Out ) .
```

Because the boundaries of the segment they recognize (*Z* and *Z*) are equal, both asserted clauses recognize an empty word (the gap). The first clause refers to gaps in only one position because the boundaries are global to the asserted clause; hence they are the same for every instance of the asserted clause. The second clause refers to an unlimited number of different gaps in different positions because the boundaries are local to the asserted clause.

So, a simple difference in quantification provides a neat formulation of the differences between several behaviours. To describe explicitly the incomplete sentences, or to handle the context in terms, blurs the differences between the behaviours. However, DCGs (even higher-order) offer nothing between their context-free grammars foundation and the Turing-complete handling of the context in Prolog. Harrop formulas offers an intermediate context handling capability that appears to offer a direct encoding of the slash feature.

Our query application (see section 6) gives us a realistic use for the mapping of GPSG rules onto Harrop formulas. A rule for generating relatives is written as follows using new grammatical connectives which will be introduced later.

```
relative M A R -->                                          % REL → pronoun S/NP
    prep_pronoun P & epsilon (formula_pr U R) &
    all i\ ( noun_phrase P A p\(p i) --> ` [] ==> sentence M (U i) ) .
```

The connectives *all* and ==> are the grammar rule versions of λProlog's *pi* and =>. Parameters *M* and *A* are respectively the mode (interrogative or declarative) and a list of agreement features. Note that agreement features are common to the pronoun and the elided noun phrase. Parameters *U* and *R* are combinators for producing the resulting semantic formula. Note the abstraction scheme: what is abstracted by *all i* is some individual. In *p\(p i)*, the abstracted individual is made into a semantic formula for noun phrases that is assumed to be the semantics of the elided noun phrase. The expression *p\(p i)* is related to the function *F* discussed

in section 6.3. In *(U i)*, the parameter U is the result of abstracting the individual over the semantics of the incomplete sentence.

Once we have adopted the idea of adding rule assumptions to DCGs, the main problem is to define what are the implicit quantifications in λHHG. In DCGs, every grammar rule or grammar item has two implicit parameters for input and output. Asserted rules of λHHG must also have the two implicit parameters. In the first rule of Pareschi and Miller's example, the input and output parameters are identical (i.e. they are connected to each other) and free in the asserted clause. In their second rule, the input and output parameters are also identical, but they are universally quantified in the assumed clause. In another case (see predicate *lambda_term* in section 5.3.2), we expect the input and output parameters are distinct and universally quantified at the level of the asserted rule (i.e. they are connected to nothing). In rule *relative* above, it is the explicit derivation into an empty string that binds the input and output parameters together. Since Pareschi and Miller present two styles of quantification and connection, and we have presented yet another, we cannot commit the translation to one of them: the quantification and connection must be flexible.

To sum up, we conceive λHHG as strongly typed DCGs plus λProlog terms and goals, plus new connectives for asserting clauses and rules, quantifying attributes in the rules, and constraining the boundaries of generated words.

5.3.2 Definition

The syntax of λHHG is given by the following rules:

$$
\begin{array}{lll}
\mathcal{G}_T & ::= \mathcal{R}_T \ .\ \mathcal{G}_T \ \mid\ \mathcal{R}_T \ . & \\
\mathcal{R}_T & ::= \mathcal{H}_T \ \text{-->}\ \mathcal{B}_T & \\
\mathcal{B}_T & ::= \mathcal{NT}_T \ \mid\ \mathcal{T}_T \ \mid\ \mathcal{B}_T \ \&\ \mathcal{B}_T \ \mid\ \mathcal{B}_T : \mathcal{B}_T \ \mid\ \mathit{epsilon}\ \mathcal{G} & \\
\mathcal{B}_T & ::= \mathit{all}\ x \backslash \mathcal{B}_T \ \mid\ \mathit{some}\ x \backslash \mathcal{B}_T & \text{\% new} \\
\mathcal{B}_T & ::= \mathit{impl}\ \ \mathcal{D}\ \ \mathcal{B}_T \ \mid\ \mathcal{R}_T \ \text{==>}\ \mathcal{B}_T & \text{\% new} \\
\mathcal{B}_T & ::= \mathit{delta}\ \Lambda_{(\mathit{list}\ T)\text{->}(\mathit{list}\ T)\text{->}o}\ \mathcal{B}_T & \text{\% new} \\
\mathcal{H}_T & ::= \mathcal{NT}_T \ \mid\ \mathcal{NT}_T\ ,\ \mathcal{T}_T & \\
\mathcal{NT}_T & ::= (\mathcal{C}_{t_1 \ldots t_i\text{->}(\mathit{list}\ T)\text{->}(\mathit{list}\ T)\text{->}o}\ \Lambda^i)_{(\mathit{list}\ T)\text{->}(\mathit{list}\ T)\text{->}o} & \\
\mathcal{T}_T & ::= \text{`}\ \Lambda_{(\mathit{list}\ T)} &
\end{array}
$$

The types of the new connectives are as follows. It is easy to check that the above grammar generates well-typed expressions.

```
type delta ((list T)->(list T)->o) ->((list T)->(list T)->o) -> ((list T)->(list T)->o) .
type (all, some) (_ -> ((list T)->(list T)->o)) -> ((list T)->(list T)->o) .
type (impl, '==>') o -> ((list T)->(list T)->o) -> ((list T)->(list T)->o) .
op 1150 xfx '==>' .
```

The first new construct that appears in the rule for generating relative clauses is an *all* quantification. A *some* quantification is designed similarly. They are universal and existential quantifications at the rule level.

Similarly, one may need clause implication at the rule level. Even if such an implied clause is to be used in condition goals only, it may be that no condition goal corresponds to its scope: it must be implied at the rule level.

The semantics of a rule body *(impl Clause Body)* is that it generates with grammar rules $\mathcal{R}$ and clauses $\mathcal{C}$ what *Body* generates with $\mathcal{R}$ and $\mathcal{C} \cup \{Clause\}$.

Rule assertion parallels implication at the grammatical level. The semantics of a rule body *Rule==>Body* is that it generates with grammar rules $\mathcal{R}$ and clauses $\mathcal{C}$ what *Body* generates with $\mathcal{R} \cup \{Rule\}$ and $\mathcal{C}$. Like every rule, an asserted rule has two implicit parameters for the boundaries of the segment its head recognizes. These parameters are universally quantified at the level of the asserted clause.

Rule assertion and the *all* quantification covers the requirement of the grammar rule for generating relatives. Another application is the construction of an higher-order abstract syntax tree. For instance, the rule for recognizing simply typed λ-terms which are abstractions (as generated by the grammar of section 3.2.1) can easily be written in λHHG.

```
lambda_term (arrow ArgType Type) (lambda E) -->
      lambda_var ArgType Var & ` "\" &
      all v\( lambda_term ArgType v --> ` Var ==> lambda_term Type (E var) ) .
```

In order to solve the problem of the quantification of an asserted rule, one must be able to capture the implicit parameters of the rule without interfering with them. One must also be able to capture the boundaries of the segment that some non-terminal or rule body expression generates. To do something special for capturing the boundaries of the segment generated by a clause head is useless because it is in the basic principles of every grammar rule that what a head generates is exactly what the corresponding body generates.

We propose the *delta* construct. The name "delta" is chosen because it connotes a difference, here between two positions in an input word.

The semantics of a rule body *(delta Cond Body)* is that it generates such words that *Body* generates and whose boundaries satisfy *Cond*. With our current typing assumption, the first argument of *delta* is any predicate of two positions. The *delta* construct relates two visions on the same input segment: the rule body expression that generates it, and a condition on its boundaries. Although the boundaries are implicit (for the sake of hiding house-keeping), the *delta* construct gives a means to express properties on them.

Another point of view is to say that the *delta* construct is to the implicit variables of grammar rules what a condition goal is to the explicit variables. It gives a means to control the instantiation of these variables. Because the explicit variables concern the attributes of the grammar (either DCG or λHHG), and the implicit variables concern the parsed/generated sentence, the *delta* construct is actually completing the capabilities of the condition goals. Though both the *delta* construct and condition goals allow for an arbitrary control on the parsing/generation process, they do not collapse the system. The other, more disciplined, constructs keep their advantage which is a very concise notation for a specialized task.

The explicit quantification in rule bodies, the implicit universal quantification at the rule level, combined with the *delta* construct, allow us to express the various quantifications/connections we have mentioned. The two examples from Pareschi and Miller's paper can now be given in a Logic Grammar formalism:

```
rel REL -->
      ` ["whom"] & some X\(all gap\( np gap --> delta in\out\(in = X) (` []) ==> s (REL gap) )) .
rel REL --> ` ["whom"] & all gap\( (np gap --> ` []) ==> s (REL gap) ) .
```

Note that the derivation to the empty terminal *(` [])* makes the input and output equal. It remains to specify their scope. The predicate *in\out\(in=X)* makes them global because variable *X* is global.

More examples can be given of constraining the boundaries of a generated word. The following rule expresses that the asserted rule *head1-->body* must be used to generate a prefix of rule body expression *b1*.

```
head0 -->
      (     head1 --> delta in\out\(in = LeftBoundary) body
      ==> ( b0 & delta in\out\(in = LeftBoundary) b1 & b2 ) ) .
```

Assuming the predicate *dlength* is a variant of the predicate *length* for difference-lists, the next rule expresses that the length of the word generated by *b1* is *Length*.

```
head Length --> b0 & delta in\out\(dlength in-out Length) b1 & b2.
```

5.3.3 The translation of λHHG into λProlog

Translating λHHG into λProlog is a mere extension of translating DCGs into λProlog. One needs only define new translation combinators and new translation clauses for the new connectives.

```
#define ALL body\in\out\(pi x\(body x in out))

#define SOME body\in\out\(sigma x\(body x in out))
```

Fill in a quantifier in a clause and specify that the input changes as the quantified goal says.

```
#define IMPL clause\body\in\out\(clause => body in out)
```

Fills in an implication in a clause and specify that the input changes as the conclusion goal says.

```
#define DELTA cond\body\in\out\(cond in out, body in out)
```

Connects two items in a parallel conjunction.

The new connectives of λHHG are defined as follows:

```
(all Body) In Out :- ALL Body In Out .
(some Body) In Out :- SOME Body In Out .
(impl Clause Body) In Out :- IMPL Clause Body In Out .
(Rule ==> Body) In Out :- dcg_rule Rule Clause , IMPL Clause Body In Out .
(delta Cond Body) In Out :- DELTA Cond Body In Out .
```

The new translation rules are as follows:

```
dcg_body (all Body0) (ALL Body) :- ! , pi x\(dcg_body (Body0 x) (Body x)) .
dcg_body (some Body0) (SOME Body) :- ! , pi x\(dcg_body (Body0 x) (Body x)) .
dcg_body (impl Clause Body0) (IMPL Clause Body) :- ! , dcg_body Body0 Body .
dcg_body (Rule ==> Body0) (IMPL Clause Body) :- ! , dcg_rule Rule Clause , dcg_body Body0 Body .
dcg_body (delta Cond Body0) (DELTA Cond Body) :- ! , dcg_body Body0 Body .
```

As a result, the λHHG rendering of Pareschi and Miller's rules translate to the following λProlog clauses.

```
rel REL In Out :-
      sigma L1\( In = ["whom" | L1] ,
      sigma X\(pi gap\
      ( (pi In\(pi Out\(np gap In Out :- In = X, In = Out))) => s (REL gap) L1 Out ) )) .
rel REL In Out :-
      sigma L1\( In = ["whom" | L1] ,
      pi gap\( (pi In\(pi Out\(np gap In Out :- In = Out))) => s (REL gap) L1 Out ) ) .
```

6 APPLICATION TO A NATURAL LANGUAGE QUERY SYSTEM

In this section, we present as an application a natural language query system for a database describing relationships between great names in the XVIIth century. The semantics of a phrase is a formula of the first-order predicate calculus represented by a simply typed λ-expression. Thus, all three possible representation for higher-order terms are usable.

The grammar of the subset of French used by the query system is given in appendix A. We show that the λ-calculus we deal with can be simply typed, which ensures strong normalization of the semantic formulas. Finally, we apply the different representations of the simply typed λ-calculus to the semantic formulas.

6.1 The grammar as λHHG rules

Every production rule is encoded by a λHHG rule. For instance,

$$\begin{array}{l} <\textit{interrogative sentence}>_{(qui\ S)} \\ \quad ::= <\textit{preposition}>_{P}\ <\textit{interrogative pronoun}>_{P,A} \\ \quad\quad \forall i\ <\textit{sentence}>_{int,(S\ i)}\ /<\textit{noun phrase}>_{P,A,\lambda p(p\ i)} \end{array}$$

is encoded into λHHG as

```
interrogative_sentence (qui S) -->
      preposition P & interrogative_pronoun P A
      all i\( noun phrase P A p\(p i) ==> sentence inter (S i) ) .
```

The parameter A is a list of *features*, for instance *fem.sing.hum* meaning that it is concerned with a feminine, singular phrase denoting a human being. It is thus possible to express agreement between the noun phrase and the verb phrase. Parameter *inter* indicates that the verb phrase is in interrogative form, involving a possible reduplication of the subject.

6.2 The semantics

The semantics of every sentence is a first-order formula represented by a simply typed λ-expression. Let us define more precisely the λ-expressions we are using.

Definition *For all types α, there is a countable set of elements called the variables of type α. The constants are the following elements:*

- Charles, Thérèse, *and more generally all the names of the lexicon, of type* i,
- homme, régner, séduisant *(man, to reign, handsome), and more generally all nouns-0, verbs-0, and adjectives-0 of the lexicon, of type* i→o,
- épouser, frère, marié *(to marry, brother, married), and more generally all verbs-1, adjectives-1, and nouns-1 of the lexicon, of type* i→i→o,
- non *(not) of type* o→o,
- et, ou *(and, or) of type* o→o→o,
- existe, tout *(exists, for all) of type* (i→o)→o.

The set Λ of the simply typed λ-expressions used as semantic formulas is the smallest set containing:

- *the variables and the constants,*
- *terms* (f a) *where* f *is in Λ with type $\alpha\rightarrow\beta$ and* a *is in Λ with type α; their type is β*
- *terms* λx(m) *where* m *is in Λ with type β and* x *is a variable with type α; their type is $\alpha\rightarrow\beta$.*

In order to represent the semantics of an input sentence, we associate to (nearly) every rule head a λ-expression which is a function of the λ-expressions associated with the syntactic items in the corresponding rule body. This function is itself represented by a λ-expression. The semantics of the input sentence is the expression attached to the non-terminal axiom. In the concrete implementation, the correspondence between syntactic items and their semantic values is established by means of an additional parameter. Thus, the computations of the semantic formula and the syntactic analysis are interleaved.

6.3 An example: "Charles épouse Thérèse"

Let us consider the sentence "Charles épouse Thérèse" (Charles marries Thérèse). Its semantics is expressed by the logical formula *épouse(Charles, Thérèse)*, (*(épouse Charles Thérèse)* in the syntax of λ-calculus). This formula results from the normalization of a more complex λ-expression, which is constructed as follows.

The word "Charles" is a name that stands as a subject phrase. The constant *Charles*, with type *i*, is associated with the word "Charles".

The expression *N* associated with the name "Charles" is the result of applying a function $F{=}\lambda i\lambda p(p\ i)$ to *Charles*. This function has type $i\rightarrow(i\rightarrow o)\rightarrow o$. Thus, *N* reduces to $\lambda p(p\ Charles)$, and its type is $(i\rightarrow o)\rightarrow o$. The type of *N* expresses the fact that a noun phrase, say "Charles", takes as argument a unary predicate, say "reigns", resulting in the proposition "Charles reigns".

The analysis of input *"Charles"* and the construction of formula $\lambda p(p\ Charles)$ are done at the same time by the λHHG rules that follows.

```
subj_n_phrase A NP --> noun_phrase subj A NP .
noun_phrase P A N --> preposition P & name A N .
preposition subj --> ` [] .
name [mas,_,_] FN --> masc_name N & epsilon (name_formula F , application F N FN) .
masc_name Charles --> ` ["Charles"] .
name_formula F :- ... .                                   % F = λiλp(p i)
```

According to the programming language and representation technique, the predicate *name_formula* that declares the shape of semantic formulas for names is one of the following:

```
name-formula(lambda(i)) ->                    % PrologII, λ-term-as-rational-tree
      eq(i, lambda(p)) eq(p, var(p).var(i)) ;
name_formula i\p\(p i) .                      % λProlog, direct
name_formula (lambda i\(lambda p\(p.i))) .    % λProlog, explicit
```

The word "épouse" (marries) is a verb. As a word it is associated with the semantic formula *épouse* of type $i \rightarrow i \rightarrow o$. Expression B, corresponding to "épouse" seen as a verb-1 is the result of applying to *épouse* a function $F' = \lambda p \lambda i \lambda j(p\ j\ i)$ of type $(i \rightarrow i \rightarrow o) \rightarrow i \rightarrow i \rightarrow o$. Thus, expression B reduces to $\lambda i \lambda j(\text{épouse}\ j\ i)$.

The word "Thérèse" is a name that stands as a complement. In the same way we did previously for the name "Charles", we associate to the *noun phrase* "Thérèse" the expression $M = \lambda p(p\ \text{Thérèse})$ with type $(i \rightarrow o) \rightarrow o$.

Let us now build the expression P corresponding to the word "Thérèse" interpreted as a *prepositional phrase*. Expression P is equal to $(F''\ M)$ where F'' is $\lambda n \lambda p \lambda i(n\ \lambda j(p\ j\ i))$. Expression F'' aims at producing the semantic formula P of a prepositional phrase given the semantic formula of a noun phrase with type $(i \rightarrow o) \rightarrow o$. The resulting semantic formula P applies to the relation defined here by the verb-1 "épouser" (to marry), and thus to an expression of type $i \rightarrow i \rightarrow o$, resulting in the unary predicate *épouser(Thérèse)* (to marry Thérèse) of type $i \rightarrow o$.

The semantic formula associated with the whole sentence is a combination of the semantic formulas of its three parts. Expression U associated with the verb phrase "épouse Thérèse" is $(P\ B)$ which is of type $i \rightarrow o$ and the expression of the whole sentence is $(N\ U)$ of type o:

$$(\lambda i \lambda p(p\ i)\quad \text{Charles}\quad (\lambda n \lambda p \lambda i(n\ \lambda i(p\ j\ i))\quad (\lambda i \lambda p(p\ i)\ \text{Thérèse})\quad (\lambda p \lambda i \lambda j(p\ j\ i)\ \text{épouse}))$$

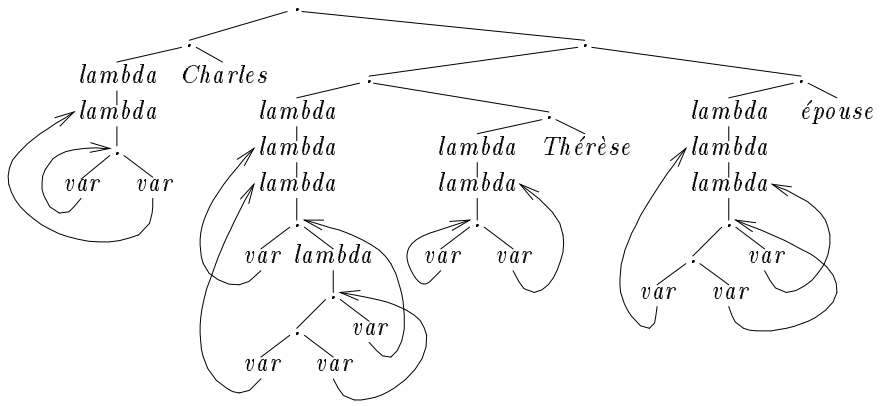
The direct and explicit λProlog representations of this λ-term are respectively:

$$(i\backslash p\backslash(p\ i)\quad \text{Charles}\quad (n\backslash p\backslash i\backslash(n\ i\backslash(p\ j\ i))\quad (i\backslash p\backslash(p\ i)\ \text{Thérèse})\quad (p\backslash i\backslash j\backslash(p\ j\ i)\ \text{épouse}))$$

and

```
( (lambda i\(lambda p\(p.i))) .Charles )
      .( ( (lambda n\(lambda p\(lambda i\(n.(lambda i\((p.j).i))))))
            .( (lambda i\(lambda p\(p.i))).Thérèse ) )
            .( (lambda p\(lambda i\(lambda j\((p.j).i)))).épouse ) )
```

The λ-term-as-rational-tree representation of this λ-term is as follows:



One can verify that they have the expected normal forms.

6.4 Types and semantic combinators

More generally, as this example shows, we associate a type with nearly every symbol of the grammar. Every λ-expression corresponding to some syntactic construct will have the type associated to that construct. Without further details, the correspondence between syntactic constructs and types is as follows:

declarative sentence	o
every noun 0, *verb, verb phrase*, *adjective 0*	$i \rightarrow o$
every noun 1, *verb 1*, *adjective 1*	$i \rightarrow i \rightarrow o$
every name, *(subject) noun phrase*	$(i \rightarrow o) \rightarrow o$
relative, *interrogative noun phrase*	$(i \rightarrow o) \rightarrow i \rightarrow o$
prepositional phrase	$(i \rightarrow i \rightarrow o) \rightarrow i \rightarrow o$
every article	$(i \rightarrow o) \rightarrow (i \rightarrow o) \rightarrow o$
quel (which?)	$(i \rightarrow o) \rightarrow (i \rightarrow o) \rightarrow i \rightarrow o$

Syntactic constructs that are not listed above are those which have no semantic value. We must mention that indefinite articles and singular definite articles are both translated into the existential quantifier, and thus are not distinguished at the semantic level. The plural definite article "les" (le) as well as generic adjectives like "tout" (every) are interpreted with the universal quantifier. Associated expressions are $\lambda p \lambda q(exist\ \lambda i(and\ (p\ i)\ (q\ i)))$ for the existential quantifier, and $\lambda p \lambda q(tout\ \lambda i(ou\ (non\ (p\ i))\ (q\ i)))$ for the universal quantifier.

6.5 Execution efficiency

Here is an example executed with the Prolog II+ compiler on a Sun 3.60.

> question: *De quel pays l'homme dont Louis XIII épouse la sœur est-il le roi ?* (Of which country is the man whose sister Louis XIII marries the king?)
>
> answer: *L'Espagne* (Spain)

In this example, the minimal system for representing the semantic formula as a rational term amounts to 27 equations. Normalization requires 0.2 seconds and displaying the answer takes 0.6 seconds. It takes roughly the same time with the explicit λProlog version on a Sun 4. The λProlog implementation used for this test is a prototype in which several well known efficient mechanisms have not yet been installed, among them clause indexing. This explains why the execution times are roughly the same as when Prolog II runs on a much slower machine. With the direct λProlog implementation, reduction time is under the measurement precision, and displaying the answer takes roughly 0.2 seconds.

7 CONCLUSION

7.1 Representing higher-order terms

Simply typed λ-calculus allows us to accurately express the semantics of natural language sentences. Typing expressions and strict type classification of the grammar symbols provide a better control of the language syntax and semantics. Thus, it is worth finding a sound and efficient representation for λ-calculus in Logic Programming.

A sound representation must provide for the management of λ-variables, and the distinction between logical variables and λ-variables. There is a whole spectrum of solutions. At one end, the sound representation can be directly provided for by the metalanguage (e.g. λProlog). At the other end, no provision is taken for such a problem (e.g. Prolog), and everything must be programmed by hand. A middle course is to use a system that

already provides for graph manipulations (Prolog II), or for λ-terms manipulations (λProlog), and to program more or less explicitly the manipulation of semantic formulas on it.

The Prolog II solution is to take advantage of rational terms for designing a nameless notation for λ-calculus, so as not to confuse Prolog variables and object-level variables. This coding is a suitable tool in Logic Programming since it avoids variable captures without requiring any renaming. It is specially convenient in the case of Prolog II or Prolog III because it benefits from the unification of rational trees which is implemented in these systems. In particular, it does not require any computation on integral indices as it is the case with De Bruijn's representation. As a consequence, the normalizer is very short and quite simple. This technique can be easily transposed to terms with coreference constraints like ψ-terms.

The second solution is to use the simply typed λ-terms of λProlog. They can be used directly, and then the λProlog system provides unification, type-checking, normalization, and substitution. They can also be used to represent explicitly abstractions and applications in a nameless fashion. In this case, the user must provide reduction, type-checking and unification modulo $\beta\eta$-equivalence (if required), but the λProlog system still provides the substitution mechanism. The explicit representation is more flexible because it can be used even when the object-level λ-terms are not the same as λProlog's terms.

Note that the three ways of implementing λ-terms in Prolog II/λProlog are very different with respect to unification. With the representations as rational trees or as explicit λ-terms, unification does not take $\beta\eta$-equivalence into account, whereas it does with the direct λProlog representation. In other words, with the first two representations, λ-equivalence is an object-level theory, but it is a metalevel theory in the last representation.

7.2 A logical handling of context

We have described a new Logic Grammar formalism called λHHG (*higher-order hereditary Harrop grammars*) that can be used in λProlog implementations.

A transposition of DCGs for λProlog is necessarily strongly typed because we want to apply the architectural rules of the implementation of DCG in Prolog to the implementation of its transposition to λProlog. As DCG obeys Prolog term syntax, its transposition obeys λProlog term syntax and typing.

To enhance attributes and conditions of DCG by higher-order terms and goals is trivial as far as the translator is concerned. To enhance rule connectives with assertions and quantifications is a deeper improvement because it gives the capability to convey context in a grammatical way. The non-grammatical way, using attributes and conditions, remains available as well.

We have shown the flexibility of λHHG for handling various situations that are described in the literature, or that we have described in this paper. λHHG has application in natural language analysis (e.g. gap location in GPSGs) as well as in programming language analysis (e.g. grammatical management of scope, analysis-time modification of the grammar).

Neither λProlog nor λHHG handles at the rule level such constraints as that some rule must be used exactly once. There are proposals for using Linear Logic Programming for solving this problem [45, 26, 25]. Our method of handling logical connectives at the grammar rule level applies to the linear connectives too.

Note that when, as usual, the strategy of interpreters for Horn clause programs is recursive-descent, the resulting DCG analyser is a poor one. However, there are bottom-up strategies that are complete for useful subsets of Horn clause programs [55]. λProlog and λHHG have the same problem, but a complete strategy is yet to be discovered, though some work is in progress [28].

7.3 Other advanced features

We have presented several ways of implementing compositional semantics in Logic Programming using rational trees and simply typed λ-terms. We have also presented a Logic Grammar formalism derived from the clause language of λProlog: higher-order hereditary Harrop grammars. More generally, we have shown how several advanced Logic Programming features, namely rational trees in Prolog II (or Prolog III), simply typed λ-terms and hereditary Harrop formulas in λProlog, can be advantageously used in the context of Computational Linguistics. Many other advanced features exist that are of interest for Computational Linguistics [51]. For instance, the ψ-terms of Login [3] can be used for manipulating frames [31], and various constraint domains can be used to model semantic domains.

The common aspect of these features is that they are integrated into Logic Programming in a smooth way. On one hand, when they deal with the computation domain (simply typed λ-terms, ψ-terms, and rational trees), they can be modeled as constraint domains. On the other hand, when they deal with the clause language (hereditary Harrop formulas, Linear Logic Programming) they can be modeled by proof theoretic systems.

These advanced features seldom live alone. First, there are necessary associations between features that make them yield their full power. For instance, simply typed λ-terms are almost useless without hereditary Harrop formulas because one needs universal quantification to "go through" abstractions (see section 3.2.3), and implication to express the properties of universal variables. Conversely, manipulating hereditary Harrop formulas as object-level formulas (i.e. metaprogramming) is more precisely done using simply typed λ-terms. The reason is the need to represent quantifications. It already existed for Horn formulas, but is more crucial for Harrop formulas because the language of quantifiers is more complex than in Prolog. As another example, for traversing rational trees without looping, one needs a sound and complete version of the negation of equality on rational terms: this is the predicate *dif* (see predicate *start-again* in section 4.1), a kind of constraint.

Second, there is a trend to integrate several features in the same system. For instance, Prolog III integrates rational trees and several other constraint domains, the Prolog/Mali [5] version of λProlog integrates both simply typed λ-terms and a restricted form of rational trees and ψ-terms. This must be done carefully to maintain the logical integrity of the systems, but it makes Logic Programming even more promising for Computational Linguistics.

References

[1] H. Abramson and V. Dahl. *Logic grammars. Symbolic computation — Artificial Intelligence*, Springer-Verlag, Berlin, FRG, 1989.

[2] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[3] H. Aït-Kaci and R. Nasr. Login: a logic programming language with built-in inheritance. *J. Logic Programming*, 3:187–215, 1986.

[4] H. Barendregt. *The Lambda Calculus: Its Syntax and Semantics.* Volume 103 of *Studies in logic and the foundations of mathematics*, North-Holland, 1981.

[5] P. Brisset and O. Ridoux. *The Compilation of λProlog and its execution with MALI.* Technical Report 1831, INRIA, 1993. ftp: //ftp.irisa.fr/local/lande.

[6] C. Brown and G. Koch, editors. *Natural Language Understanding and Logic Programming.* North-Holland Elsevier, Amsterdam, 1980.

[7] A. Church. A formulation of the simple theory of types. *J. Symbolic Logic*, 5(1):56–68, 1940.

[8] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer-Verlag, 1987.

[9] A. Colmerauer. An interesting subset of natural language. In K.L. Clark and S-Å. Tärnlund, editors, *Logic Programming*, pages 45–66, London Academic Press, 1982. (APIC Studies in Data Processing 16).

[10] A. Colmerauer. An introduction to Prolog III. *CACM*, 33(7), 1990.

[11] A. Colmerauer. *Les systèmes-Q ou un formalisme pour analyser et synthétiser des phrases sur ordinateur.* Publication Interne 43, Département d'Informatique, Université de Montréal, Canada, 1970.

[12] A. Colmerauer. Metamorphosis grammars. In L. Bolc, editor, *Natural Language Communication with Computers*, pages 133–187, Springer-Verlag, 1978.

[13] A. Colmerauer. Prolog and infinite trees. In K.L. Clark and S-Å. Tärnlund, editors, *Logic Programming*, pages 231–251, Academic Press, New-York, 1982.

[14] A. Colmerauer, H. Kanoui, and M. Van Caneghem. Prolog, bases théoriques et développements actuels. *TSI*, 2(4), 1982.

[15] S. Coupet-Grimal. *Deux arguments pour les arbres infinis en Prolog.* Thèse, Université d' Aix-Marseille 2, 1988.

[16] S. Coupet-Grimal. Prolog infinite trees and automata. *RAIRO Informatique Théorique et Applications*, 25(5):397–418, 1991.

[17] V. Dahl. Translating spanish into logic through logic. *American J. Computational Linguistics*, 7(3):149–164, 1981.

[18] V. Dahl and P. Saint-Dizier, editors. *Proc. 1st Int. Work. Natural Language Understanding and Logic Programming.* North-Holland, Amsterdam, 1985.

[19] V. Dahl and P. Saint-Dizier, editors. *Proc. 2nd Int. Work. Natural Language Understanding and Logic Programming.* North-Holland, Amsterdam, 1988.

[20] N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[21] A. Gal, G. Lapalme, P. Saint-Dizier, and H. Somers. *Prolog for Natural Language Processing.* Wiley, 1992.

[22] G. Gazdar, E. Klein, G.K. Pullum, and I.A. Sag. *Generalized Phrase Structure Grammar.* Basil Blackwell, 1985.

[23] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, 50:1–102, 1987.

[24] J. Haas and B. Jayaraman. Interactive synthesis of definite-clause grammars. In K. Apt, editor, *Joint Int. Conf. and Symp. Logic Programming*, pages 541–555, MIT Press, 1992.

[25] J.S. Hodas. *Logic Programming in Intuitionistic Linear Logic: Theory, Design, and Implementation.* PhD. Thesis, University of Pennsylvania, Department of Computer and Information Science, 1993.

[26] J.S. Hodas. Specifying filler-gap dependency parsers in a linear-logic programming language. In K. Apt, editor, *Joint Int. Conf. and Symp. Logic Programming*, pages 622–636, MIT Press, 1992.

[27] J.S. Hodas and D.A. Miller. Logic programming in a fragment of intuitionistic linear logic. *Information and Computation*, 110(2):327–365, 1994.

[28] A. Hui Bon Hoa. Some kind of magic for (a restriction of) L_λ. In *Workshop on λProlog*, Philadelphia, PA, USA, 1992.

[29] S. Le Huitouze, P. Louvet, and O. Ridoux. Logic grammars and λProlog. In D.S. Warren, editor, *10th Int. Conf. Logic Programming*, pages 64–79, MIT Press, 1993.

[30] S.C. Johnson. *YACC — yet another compiler compiler.* Computing Science Tech. Rep. 32, Bell Laboratories, Murray Hill, NJ, 1978.

[31] K. Knight. Unification: a multidisciplinary survey. *ACM Computing Surveys*, 21(1):93–124, 1989.

[32] R. Kowalski and M. Van Emden. The semantics of predicate logic as a programming language. *JACM*, 23(4):733–743, Oct. 1976.

[33] T.K. Lakshman and U.S. Reddy. Typed Prolog: a semantic reconstruction of the Mycroft-O'Keefe type system. In *Int. Logic Programming Symp.*, pages 202–217, 1991.

[34] J.W. Lloyd. *Foundations of Logic Programming. Symbolic computation — Artificial Intelligence*, Springer-Verlag, Berlin, FRG, 1987.

[35] M. McCord. Using slots and modifiers in logic grammars for natural languages. *Artificial Intelligence*, 18:327–368, 1982.

[36] D. Miller and G. Nadathur. Some uses of higher-order logic in computational linguistics. In *24st Annual Meeting of the Association for Computational Linguistics*, pages 247–255, 1986.

[37] D.A. Miller and G. Nadathur. Higher-order logic programming. In E. Shapiro, editor, *3rd Int. Conf. Logic Programming, LNCS 225*, pages 448–462, Springer-Verlag, 1986.

[38] D.A. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[39] D.A. Miller, G. Nadathur, and A. Scedrov. Hereditary Harrop formulas and uniform proof systems. In D. Gries, editor, *2nd Symp. Logic in Computer Science*, pages 98–105, Ithaca, New York, USA, 1987.

[40] R. Milner. A theory of type polymorphism in programming. *J. Computer and System Sciences*, 17:348–375, 1978.

[41] R. Montague. *Formal Philosophy.* Yale University Press, New Haven, Co, USA, 1974.

[42] A. Mycroft and R.A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[43] R.A. O'Keefe. *The Craft of Prolog.* MIT Press, 1990.

[44] R. Pareschi. A definite clause version of categorial grammar. In *26th Annual Meeting of the Association for Computational Linguistics*, pages 270–277, 1988.

[45] R. Pareschi and D.A. Miller. Extending definite clause grammars with scoping constructs. In D.H.D. Warren and P. Szeredi, editors, *7th Int. Conf. Logic Programming*, pages 373–389, MIT Press, 1990.

[46] F.C.N. Pereira. *Logic for Natural Language Analysis.* Technical Note 275, SRI International, Stanford, Ca, 1983.

[47] F.C.N. Pereira. Prolog and natural-language analysis: into the third decade. In S. Debray and M. Hermenegildo, editors, *2nd North American Conf. Logic Programming*, pages 813–832, MIT Press, 1990.

[48] F.C.N. Pereira and D.H.D. Warren. Definite clauses for language analysis. *Artificial Intelligence*, 13:231–278, 1980.

[49] J.-F. Pique. On a semantic representation of natural language sentences. In M. Van Caneghem, editor, *1st Int. Conf. Logic Programming*, pages 215–223, Marseille, France, 1982.

[50] J.A. Robinson. A machine-oriented logic based on the resolution principle. *JACM*, 12(1):23–41, 1965.

[51] P. Saint-Dizier. *Advanced Logic Programming for Language Processing.* Academic Press, 1994.

[52] P. Saint-Dizier. An approach to natural language semantics in logic programming. *J. Logic Programming*, 3:329–356, 1986.

[53] N. Shankar. *A Mechanical Proof of the Church-Rosser Theorem.* Technical Report 78712, Institute for Computing Science, The University of Texas at Austin, 1985.

[54] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.

[55] D.S. Warren. Memoing for logic programs. *CACM*, 35(3):94–111, 1992.

[56] D.S. Warren. Using λ-calculus to represent meanings in logic grammars. In *21st Annual Meeting of the Association for Computational Linguistics*, pages 51–56, Cambridge, Ma, USA, 1983.

A THE GRAMMAR

Follows an attributed context-free grammar, augmented with the *slash* construct and universal quantification in rule bodies, for defining the subset of French used in the query application. Terminal symbols are in boldface and the axioms are is *declarative sentence* and *interrogative sentence*. Attributes are subscripted and separated by commas. An attribute controls mode (*decl*/*int* for declarative or interrogative); its generic name is M. An attribute controls agreement of the rôle assigned by prepositions (*sub*/*dir*/*de*/*a*); its generic name is P or P'. Another attribute is a set of agreement features (*sing*/*plur* for singular or plural, *fem*/*mas* for feminine or masculine, and *hum*/*n-hum* for human or non-human); its generic name is A or A'. Unification of two sets fails if they contain incompatible features. When it succeeds the solution is the union of the two attributes. Semantic attributes are called after the first letters of the non-terminal that synthesize them.

$<\mathit{declarative\ sentence}>_S$ ::= $<\mathit{sentence}>_{decl,S}$ **.**
$<\mathit{interrogative\ sentence}>_S$ ::= $<\mathit{sentence}>_{int,S}$ **?**
$<\mathit{interrogative\ sentence}>_{(qui\ VP)}$
::= **qui** $<\mathit{verb\ phrase}>_{decl,\{sing,hum\},VP}$ (who ... (?))

| **qu'est-ce-qui** $<verb\ phrase>_{decl,\{sing,n-hum\},vp}$ (what ... (?))

$<interrogative\ sentence>_{(qui\ S)}$
::= $<preposition>_P$ $<interrogative\ pronoun>_{P,A}$ $\forall i$ $<sentence>_{int,(S\ i)}/<noun\ phrase>_{P,A,\lambda p(p\ i)}$

$<interrogative\ sentence>_{(qui\ (PP\ N))}$ ::= $<quel\ est\ le>_A$ $<noun\ 1>_{A,P,A',N}$ $<prepositional\ phrase>_{P,A',PP}$

$<interrogative\ sentence>_{(qui\ (REL\ N))}$ ::= $<quel\ est\ le>_A$ $<noun\ 0>_{N,A}$ $<relative>_{decl,A,REL}$

$<interrogative\ sentence>_{(qui\ (INP\ S))}$
::= $<preposition>_P$ $<interrogative\ noun\ phrase>_{A,INP}$
$\forall i$ $<sentence>_{int,(S\ i)}/<noun\ phrase>_{P,A,\lambda p(p\ i)}$

$<interrogative\ sentence>_{(est-ce-que\ S)}$ ::= **est-ce-que** $<declarative\ sentence>_S$ (is ... (?))

$<interrogative\ sentence>_{(est-ce-que\ (SNP\ VP))}$ ::= $<subject\ noun\ phrase>_{A,SNP}$ $<verb\ phrase>_{int,A,VP}$

$<sentence>_{M,(SNP\ VP)}$ ::= $<subject\ noun\ phrase>_{A,SNP}$ $<verb\ phrase>_{M,A,VP}$

$<subject\ noun\ phrase>_{A,NP}$ ::= $<noun\ phrase>_{sub,A,NP}$

$<prepositional\ phrase>_{P,A,(\lambda p(p\ NP))}$ ::= $<noun\ phrase>_{P,A,NP}$ *if P≠suj*

$<noun\ phrase>_{P,A,N}$ ::= $<preposition>_P$ $<name>_{A,N}$

$<noun\ phrase>_{P,A,(PA\ (REL\ N))}$ ::= $<preposition\ article>_{P,A,PA}$ $<noun\ 0>_{A,N}$ $<relative>_{decl,A,REL}$

$<noun\ phrase>_{P,A,(PA\ (PP\ N))}$
::= $<preposition\ article>_{P,A,PA}$ $<noun\ 1>_{A,P',A',N}$ $<prepositional\ phrase>_{P',A',PP}$

$<preposition\ article>_{de,\{mas,sing\},\lambda p\lambda q\exists i(q\ i\wedge p\ i)}$ ::= **du** (of the)

$<preposition\ article>_{de,\{plur\},\lambda p\lambda q\forall i(q\ i\Rightarrow p\ i)}$ ::= **des**

$<preposition\ article>_{a,\{masc,sing\},\lambda p\lambda q\exists i(q\ i\wedge p\ i)}$ ::= **au** (to the)

$<preposition\ article>_{a,\{plur\},\lambda p\lambda q\forall i(q\ i\Rightarrow p\ i)}$ ::= **aux**

$<preposition\ article>_{P,A,ART}$ ::= $<preposition>_P$ $<article>_{A,ART}$

$<preposition>_{de}$ ::= **de** | **d'** (of)

$<preposition>_a$ ::= **à** (to)

$<preposition>_A$ ::= ϵ *A=dir or A=sub*

$<article>_{A,ART}$ ::= $<indefinite\ article>_{A,ART}$ | $<definite\ article>_{A,ART}$ | $<other>_{A,ART}$

$<indefinite\ article>_{\{mas,sing\},\lambda p\lambda q\exists i(q\ i\wedge p\ i)}$ ::= **un** (a)

$<indefinite\ article>_{\{fem,sing\},\lambda p\lambda q\exists i(q\ i\wedge p\ i)}$ ::= **une** (a)

$<indefinite\ article>_{\{plur\},\lambda p\lambda q\exists i(q\ i\wedge p\ i)}$ ::= **des**

$<definite\ article>_{\{mas,sing\},\lambda p\lambda q\exists i(q\ i\wedge p\ i)}$ ::= **le** | **l'** (the)

$<definite\ article>_{\{fem,sing\},\lambda p\lambda q\exists i(q\ i\wedge p\ i)}$ ::= **la** | **l'** (the)

$<definite\ article>_{\{plur\},\lambda p\lambda q\forall i(q\ i\Rightarrow p\ i)}$ ::= **les** (the)

$<other>_{\{mas,sing\},\lambda p\lambda q\forall i(q\ i\Rightarrow p\ i)}$ ::= **tout** (all)

$<other>_{\{fem,sing\},\lambda p\lambda q\forall i(q\ i\Rightarrow p\ i)}$ ::= **toute** (all)

$<other>_{\{sing\},\lambda p\lambda q\forall i(q\ i\Rightarrow p\ i)}$ ::= **chaque** (every)

$<verb\ phrase>_{M,A,ATTR}$ ::= $<verb\ to\ be>_A$ $<reduplication>_{M,A}$ $<attribute>_{A,ATTR}$

$<verb\ phrase>_{M,A,V}$ ::= $<verb\ 0>_{A,V}$ $<reduplication>_{M,A}$

$<verb\ phrase>_{M,A,(GP\ V)}$ ::= $<verb\ 1>_{A,P,A',V}$ $<reduplication>_{M,A}$ $<prepositional\ phrase>_{P,A',GP}$

$<reduplication>_{decl,\{\}}$ ::= ϵ

$<reduplication>_{int,\{mas,sing\}}$ ::= **-t-il** | **-il**

$<reduplication>_{int,\{mas,plur\}}$ ::= **-t-ils** | **-ils**

$<reduplication>_{int,\{fem,sing\}}$ ::= **-t-elle** | **-elle**

$<reduplication>_{int,\{fem,plur\}}$ ::= **-t-elles** | **-elles**

$<attribute>_{A,ADJ}$ ::= $<adjective\ 0>_{A,ADJ}$

$<attribute>_{A,(PP\ ADJ)}$ ::= $<adjective\ 1>_{A,P,A',ADJ}$ $<prepositional\ phrase>_{P,A',PP}$

$<attribute>_{A,N}$ ::= $<indefinite\ article>_A$ $<noun\ 0>_{A,N}$

$<attribute>_{A,(PP\ N)}$ ::= $<definite\ article>_A$ $<noun\ 1>_{A,P,A',N}$ $<prepositional\ phrase>_{P,A',PP}$

$<name>_{\{fem\},\lambda p(p\ N)}$ ::= $<feminine\ name>_N$

$<name>_{\{mas\},\lambda p(p\ N)}$ ::= $<masculine\ name>_N$

$<noun\ 0>_{\{fem,A\},N}$ ::= $<feminine\ noun\ 0>_{A,N}$

$<noun\ 0>_{\{mas,A\},N}$ ::= $<masculine\ noun\ 0>_{A,N}$

$<noun\ 1>_{\{fem,A\},P,A',N}$ ::= $<feminine\ noun\ 1>_{A,P,A',N}$

$<noun\ 1>_{\{mas,A\},P,A',N}$ ::= $<masculine\ noun\ 1>_{A,P,A',N}$

$<relative>_{M,A,\lambda p\lambda i(p\ i\wedge S\ i)}$ ::= $<preposition\ pronoun>_P$ $\forall i$ $<sentence>_{M,(S\ i)}/<noun\ phrase>_{P,A,\lambda p(p\ i)}$

$<relative>_{decl,\{\},\lambda x(x)}$ ::= ϵ

$<preposition\ pronoun>_{de}$ ::= **dont** (whose)

$<preposition\ pronoun>_a$ ::= **à qui** (whom)

$<preposition\ pronoun>_{dir}$ ::= **que** (that)
$<preposition\ pronoun>_{suj}$ ::= **qui** (who)
$<interrogative\ noun\ phrase>_{A,(Q\ N)}$ ::= $<quel>_{A,Q}$ $<noun\ 0>_{A,N}$
$<interrogative\ noun\ phrase>_{A,(Q\ (PP\ N))}$ ::= $<quel>_{A,Q}$ $<noun\ 1>_{A,P,A',N}$ $<prepositional\ phrase>_{P,A',PP}$
$<interrogative\ noun\ phrase>_{A,\lambda p\lambda i(ART\ (N\ i)\ p)}$
::= $<definite\ article>_{A,ART}$ $<noun\ 1>_{A,P,A',N}$ $<preposition>_{P}$ $<interrogative\ pronoun>_{P,A'}$
$<quel\ est\ le>_{A}$ ::= $<quel>_{A}$ $<verb\ to\ be>_{A}$ $<definite\ article>_{A}$
$<interrogative\ pronoun>_{P,\{hum\}}$ ::= **qui** (who)
$<interrogative\ pronoun>_{dir,\{n-hum\}}$ ::= **que** (what)
$<interrogative\ pronoun>_{P,\{n-hum\}}$ ::= **quoi** *if* $P \neq dir$, (what)
$<quel>_{\{mas,sing\},\lambda p\lambda q\lambda i(q\ i \wedge p\ i)}$ ::= **quel** (which)
$<quel>_{\{fem,sing\},\lambda p\lambda q\lambda i(q\ i \wedge p\ i)}$ ::= **quelle**
$<quel>_{\{mas,plur\},\lambda p\lambda q\lambda i(q\ i \wedge p\ i)}$ ::= **quels**
$<quel>_{\{fem,plur\},\lambda p\lambda q\lambda i(q\ i \wedge p\ i)}$ ::= **quelles**
$<verb\ to\ be>_{\{sing\}}$ ::= **est** (is)
$<verb\ to\ be>_{\{plur\}}$ ::= **sont** (are)
$<masculine\ name>_{Charles}$ ::= **Charles** | ...
$<feminine\ name>_{Therese}$ ::= **Thérèse** | ...
$<masculine\ noun\ 0>_{\{sing,hum\},ministre}$ ::= **ministre** | ... (minister, ...)
$<feminine\ noun\ 0>_{\{sing,hum\},reine}$ ::= **reine** | ... (queen, ...)
$<masculine\ noun\ 1>_{\{hum,sing\},de,\{hum\},frere}$ ::= **frère** | ... (brother (of), ...)
$<feminine\ noun\ 1>_{\{hum,sing\},de,\{hum\},soeur}$ ::= **soeur** | ... (sister (of), ...)
$<verb\ 0>_{\{hum,sing\},regner}$ ::= **régne** | ... (reigns, ...)
$<verb\ 1>_{\{hum,sing\},dir,\{hum\},epouser}$ ::= **épouse** | ... (marries, ...)
$<adjective\ 0>_{\{hum,sing,masc\},seduisant}$ ::= **séduisant** | ... (handsome, ...)
$<adjective\ 1>_{\{hum,sing,masc\},a,\{hum\},marie}$ ::= **marié** | ... (married (to), ...)

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399