Presentation at the 4th International Pragmatics Conference, Kobe, Japan, July 25–30, 1993.

Why should grammars (and other things) not adapt themselves to context and discourse?

Henning Christiansen
Roskilde University, Computer Science Dept.[1]
P.O. Box 260
DK-4000 Roskilde, Denmark
E-mail: henning@dat.ruc.dk

Abstract. New methods from meta-programming in logic makes it possible automatically to extract grammar rules and other formal descriptions from a given text which characterize its inherent context and discourse. As context and discourse often vary within a text it is also suggested that different grammars be applied to different portions of the text and similarly for other relevant, formal descriptions.

1 Motivation

Logic programming languages such as Prolog have proved their relevance for formal descriptions of aspects of natural language. A logic program has (in principle) a declarative reading as a logical specification and at the same time, it may serve as an efficient device for automatic text analysis. A formal description of a language will typically consist of a grammar (showing case, gender, number, etc.) and additional extra-grammatical information concerned with semantics and pragmatics. Several methods for representing grammars by logical clauses has been suggested, definite clause grammars (Pereira, Warren, 1980) are probably the most well-known. Gazdar and Mellish (1989) give an overview of various extra-grammatical representations in logic programs as well as of other sorts of logical grammars.

However, the typical logic programming approach is not so well suited when it comes to the important aspects of context and discourse, which are concerned with phenomena that are specific, not only for a group texts or a single text, but typically varies within a text. A scientific report, for example, refers to a very specific context and will often define and use its own strange notions and linguistic forms. There are often also technical appendices which only give sense for a very small minority of readers (e.g. of linguists that are experienced Prolog programmers). The problem is that a logic program is static in the sense that it contains a fixed selection of clauses. In the linguistic setting this means that a fixed and unchangeable grammar must be assumed and similarly for the extra-grammatical parts. In this sense, one can only deal with texts that can be

[1] Author's participation in the conference supported by Centre for Cognitive Science, Roskilde University.

immediately explained from the knowledge one already has — and such texts would really classify to any definition of the word “boring”.

New methods developed in the field of meta-programming in logic give a much more flexible treatment of logic programs which seems appropriate when approaching these problems. In the first place, we can explicitly state that different programs be applied to different parts of a text, e.g., by extending or restricting a given grammar. Secondly, we can leave parts of the program *unspecified* and leave it up to the program interpreter to complete it in order to successfully finish the computation.

The interpreter may — in principle and if used properly — generate grammar rules from unusual language usages or generate the necessary addenda to an existing encyclopedia in order to explain the contents of a given text.

In the following we will informally present a logic language suitable for our purpose and demonstrate its application to natural language analysis by a few simplified examples.

2 A logical meta-programming language

Our language generalizes Prolog (e.g., Clocksin, Mellish, 1987) in that each goal to be proved carries a representation of the program in which it is expected to be proved. We may, for example, define a collection of clauses as a named program which, then, can be referred to in the queries given to our interpreter. If `pppp` is the name of a program we can pose the following query to the interpreter.

```
?- p(X) in pppp.
```

The interpreter will, if possible, answer with a value for the variable `X` which makes `p(X)` provable in the program `pppp`. This is opposed to a Prolog system, in which only one program, i.e., the one in system’s memory, is available at a time. Another radical difference is that our interpreter can answer queries with variables which stands for unknown programs or program parts. The following query,

```
?- p(a) in X.
```

means intuitively “give me a program in which `p(a)` is provable” and the interpreter can in principle generate, as values of `X`, a representation of any program in which `p(a)` is provable. In most cases, however, we are of course not interested in *all* such programs but only those that belong to a certain class. In a linguistic application we may perhaps be interested in programs that can be understood as a specific kind of grammars. In order to express such requirements, we have to write a program, say with the name `metaProgram`, which defines a predicate `permissible(-)` and use in the query as follows.

```
?- permissible(X) in metaProgram, p(a) in X.
```

The interpreter will, if possible, generate values of `X` which satisfy the `permissible` predicate as well as make `p(a)` provable. — How programs actually are represented and how the interpreter works are not relevant for this presentation; interested readers are referred to (Christiansen, 1990a, 1992). We will only mention that our interpreter includes a co-routine mechanism which provides an interleaved execution of, e.g., the two goals in the query above (Christiansen, 1993a). It should also be emphasized that the novel facilities does not depend on any complicated and hopelessly inefficient "intelligent" techniques; they can actually be merged into to an existing Prolog system without any significant slowdown (Christiansen, 1993b).

The clauses in our language look like clauses in Prolog except that they may apply the "`in`" notation (as shown above) in order to change the particular program in which each goal executes. This will be explained when it appears in the examples.

3 Automatic extraction of grammars and other things

The following toy example illustrates how a partially specified language description can extend automatically in order to suit the context inherent in a given text. We will consider a language of small stories put together from small sentences such as "Fido bites Peter". Assume that `initial_grammar` is a program consisting of the following clauses; for simplicity, we will use the definite clause grammars notation [2] (Pereira, Warren, 1980).

```
story → sentence, more_story.
more_story → ['.'].
more_story →  [';'], story.

sentence →
      subject(SubjectCategory),
      verb(WhichVerb),
      object(ObjectCategory),
      {role(WhichVerb, SubjectCategory, ObjectCategory)}.

subject(Category) → name(Category).
object(Category) → name(Category).

verb(kicks) → [kicks].
verb(bites) → [bites].
verb(kisses) → [kisses].

name(man) → [peter].
name(woman) → [karen].
name(dog) → [fido].
```

The first three rules state collectively that a story is a sequence of sentences separated by semicolon and the whole story terminated by a period. The next rule defines the overall shape of a sentence. Each subject and object belongs to some category, and the `role` condition must hold for the actual verb and these categories.

---

[2] There is a simple translation of grammar rules into logic clauses. The first rule, for example, is translated into `story(S1,S3):- sentence(S1,S2), more_story(S2,S3)`. Things in curly brackets, as in the fourth clause, is not affected by the translation.

The possible application of verbs — or rather those applications foreseen by the editor of the language description — is defined by a program `initial_roles` which consists of the following clauses.

```
role(kicks, woman, dog).
role(kicks, man, dog).
role(bites, dog, woman).
role(bites, dog, man).
role(kisses, woman, man).
role(kisses, man, woman).
```

Our interpreter will now correctly answer `yes` when given a query with a "correct" sentence as follows.[3] The ampersand combine the clauses of two programs into one program.

```
?- story([peter, kisses, karen, ';', fido, bites, peter, '.'], [])
         in initial_grammar & initial_roles
```

If we analyze another story `[peter, kisses, mary, '.']` according to `initial_grammar & initial_roles` our interpreter will correctly answer `no` since it does not know anything about `mary`. Intuitively this is not very satisfactory. For a human reader, it is clear what goes on. A new character named `mary` shows up on the stage — and since we know what sorts of objects that usually are relevant for which verbs, we may conclude that the new character must be a `woman`. In order to have the interpreter to behave in a similar way, we should use its ability to generate new programs. We will extend the program expression into `initial_grammar & initial_roles & NP` where `NP` is a variable which thus stands for an unknown program, i.e., the interpreter can use the rules of the initial programs and, if they are insufficient, invent some new ones. However, the interpreter is only a poor mechanical device, industrious but without intuition, so we must give it a hint about which sorts of rules to generate. In this example, the interesting rules are those of the form

```
name(...) → [...].
```

i.e., rules similar to those for `peter`, `karen`, and `fido`. Assume that we have written another program `namesMeta` which formalizes this through a predicate `name_program`. Now, given the query,

```
?- name_program(NP) in namesMeta,
   story([peter, kisses, mary, '.'], [])
      in initial_grammar & initial_roles & NP
```

the interpreter will naturally produce an answer which states that `NP` must represent a program which contains the following rule.

---

[3] The additional empty list in the `story` predicate has to do with the translation of definite clause grammars to program clauses, cf. the previous footnote.

```
name(woman) → [mary].
```

In a similar way, we can also allow the interpreter to introduce new role and verb rules. Given the story `[peter, xxxes, karen, ';', peter, xxxes, julie, '.']`, it will from the first sentence recognize a new verb together with a role description stating that men can `xxx` women. The second sentence implies either that `julie` is a `woman` or, alternatively, that `julie` belongs to some other category for which there should be generated a `role` relation (`julie` might be a `dog` or a `penguin` in which case `dog`s or, respectively, `penguin`s would be `xxx`able). We may continue allowing the creation of new categories, e.g., from sentences of the form `[julie, is, a, giraffe]`, or trying to have the interpreter to generate grammar rules for other sentence patterns which were not foreseen in the initial grammar.

The real challenge in using this technology is to learn how to formalize the conditions that define the sort of grammars and extra-grammatical descriptions we want (by means of programs such as those above whose names carry the suffix -`Meta`). Too weak conditions, for example, can result in the generation of a terrible amount of rubbish programs. — Most linguists probably know what it means for a grammar to be a *good* grammar for a certain text corpus, but it is another and difficult matter to formalize this knowledge into an unambiguous, logical predicate.

In its absolute consequence, we may use the principle for deciphering as in this fictitious example.

```
good_grammar(G) in grammarMeta,
resembles(G, punic) in compareGrammars,
good_dictionary(D) in dictionaryMeta,
good_encyclopedia(E) in encyclopediaMeta,
text([❚❈▼, ▼❊❇▲, ✲❈, ❍❁❄■❈▲▲,  ...], []) in G & D & E,
text([▼❊❄❒❈, ❇▲, ❍❈▼❊❑❈, ❇■, ❇▼,  ...], []) in G & D & E,
...
```

4 Applying different grammars (and other things) to different parts of a text

As we saw above, meta-programming techniques can be used for having a language description in the shape of a logic program automatically adapt to suit a given text. Here we will extend this by allowing different programs to be applied to different portions of a text. We will illustrate the principle by an example from the syntax of programming languages taken from (Christiansen, 1990b). Programming languages are very convenient for illustrating the notion of context due to their explicitly defined scope rules. The following logical grammar rule describes a block construct with local declarations.

```
block in GlobalContext →
          [declare], declarations(NewRules),
          [begin],
                statements in GlobalContext & NewRules,
          [end].
```

The variable `GlobalContext` will by definition of our logic language, when this rule is applied, be bound to a program which captures all available constructs in the surrounding scope. The declarations, when analyzed, will produce a set of new rules (for new variables, procedures, types, whatever) referred to by the variable `NewRules`. Now the inner scope is analyzed using the combination of clauses `GlobalContext & NewRules`.

This example is analogous to the phenomenon we discussed in the introduction for a scientific report in which the technical appendices may use a language different from the rest of the report. There is also a clear parallel to epic texts; here any description related to context and discourse must change throughout the text (otherwise the text would not be epic). A mathematical textbook resembles also the programming language example above, with its strictly ordered definitions of new concepts and language usages.

5 Perspectives

We have shown how meta-programming techniques add a dynamic dimension to formal description and analysis of natural languages. It seems especially relevant for phenomena concerned with context and discourse. It should be emphasized that we do not wish to advocate any particular way of representing grammars and extra-grammatical information in logic. Our message is rather that out methods may add the desired dynamic properties to such approaches. However, in order to prove the usefulness of our proposal, obviously more serious examples that those shown in the present paper must be investigated. — [4]Our approach to meta-programming in logic differs from most others in that we avoid an explicit meta-interpreter and instead integrate the notions in the logical language. In this way we retain a simple logical semantics and achieve a logical completeness which implies the power to generate programs automatically.

[4] The reader may notice that this dash indicates a change in context. The word "semantics", for example, refers here to the computer scientists' notion of that name, which is different the linguists' notion of the same name which we have used elsewhere in this paper.

References

Christiansen, H. Declarative semantics of a meta-programming language, *META90, Workshop on Meta-Programming in Logic, 4–6 April 1990, Leuven, Belgium*, pp. 150–168, 1990a.

—, A survey of adaptable grammars, *SIGPLAN Notices*, 25/11, pp. 35–44, 1990b.

—, A complete resolution principle for logical meta-programming languages, META92, Workshop on Meta-Programming in Logic, 10–12 June 1992, Uppsala, Sweden, 1992. *Lecture Notes in Computer Science* 649, pp.205–219, Springer-Verlag 1992.

—, A complete demo predicate with co-routine control for automatic program synthesis, Extended abstract presented at *Second Compulog Area Meeting on Programming Languages joint with Workshop on Logic Languages, Progretto Finalizzato Sistemi Informatici e Calcolo Parallelo*, Pisa, Italy, May 6–7,1993a.

—, Mapping meta-level computations effectively to the base level. *To appear*, 1993b.

Clocksin, W.F. and Mellish, C.S., *Programming in Prolog, Third, Revised and Extended Edition.* Springer-Verlag, 1987.

Gazdar, G. and Mellish, C., *Natural language processing in Prolog, An introduction to computational linguistics*, Addison-Wesley,1989.

Pereira, F.C.N. and Warren D.H.D., Definite clause grammars for languages analysis — A survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence* 13, pp. 231–278, 1980.