A Higher Order Logic Parser for Natural Language Implemented in Lambda Prolog

by

Per Kreuger

SICS R88008
Research Report
ISSN 0283-3638

A Higher Order Logic Parser for Natural Language Implemented in Lambda Prolog.

Per Kreuger

Swedish Institute of Computer Science
Box 1263
S-164 28 KISTA
Sweden

Keywords:

Higer Order Logic, Montague Sematics, Natural Language Interpretation.

Abstract:

This paper describes an implementation of some of the ideas presented by F. C. N. Pereira in [1]. Pereira uses a sequent-calculus like system to produce Montague semantics from natural language. The lefthand sequent can be interpreted as a set of constraints under which a given sentence fragment has a certain interpretation. Pereira presents a few complementary "discharge-rules" to reduce the number of such constraints. These conditional interpretations constitute a uniform way to represent partial knowledge during the parsing process.

The implementation this paper describes is done in Lambda Prolog [2]. Lambda Prolog is a generalization of horn-clause logic to higher order logic, based on the higher order unification procedure of Huet [3]. It appears that the implementation of Pereira´s system in Lambda Prolog becomes very natural. The higher order unification mechanism of Lambda Prolog takes care of the complicated binding mechanisms in Pereira´s "discharge-rules" in a very simple and elegant way.

Finally the paper discusses some problems with the implementation and gives a few suggestions on how these could be overcome.

1. A short review of Pereiras ideas.

In his paper *Towards a Deductive Theory of Sentence Interpretation,* Pereira suggests a method to avoid the limitation of strict compositionality in semantic interpretation of natural language. The basic idea is to use an intermediate representation, that has a compositional nature, and special functions (or rather relations) mapping this intermediate representation into its logical form, i.e. Montague semantics.

A problem with compositional semantics is lack of information during the translation process. If the interpretation could be made conditional, this lack of information would not be fatal. We could make this conditional interpretation our intermediate representation. It would not represent the interpretation of the sentence but rather be a constraint on the possible interpretations of the sentence.

Pereira partitions the rules of a grammar into two kinds: 1) structural rules, that are strictly compositional, 2) discharge rules, that apply when certain types of conditions occur in the condition part of the conditional interpretation.

Pereira uses a system similar to sequent calculus to encode the rules of his grammar. This has the drawback that the discharge rules have to be complemented by supplementary constraints on the variables occurring in the conditional interpretations. This is, as we shall see, a mayor source of difficulties of this implementation as well.

The idea of conditional interpretations and the simplicity with which it is used in Pereira´s system seems to me a very promising line of investigation.

2. Lambda prolog.

Lambda Prolog is a logic programming language based on a generalization of horn-clause logic to higher order logic. It uses a special form or resolution [4] and the higher order unification procedure of Huet [3]. The syntax of Lambda Prolog permits abstraction and application in the manner of a strongly typed λ-calculus. This gives us the possibility to compute and use real functions in a logic programming framework. Of course, in many cases this leads to intractable problems, but Lambda Prolog seems to prove itself in more and more applications in spite of its fundamental incompleteness.

I have no space here to go further into a presentation of the language itself. I recommend interested readers to get Miller´s & Nadathur´s presentation of the language [2], or one of several other papers they published, describing various applications of the language.

I will give a short description of the syntax of the language to aid the reader in following the examples and pieces of code present in this paper.

Lambda Prolog uses function application and abstraction. The syntax of application is simply a sequence of terms, i.e."`f a b`" represents the result of applying the function `f` to its arguments `a` and `b`. Syntactically predicates are just a special case of functions (those returning boolean values), so the the call of a predicate looks just like a function application.

The syntax of abstraction uses the \ operator. λx.g(x) is represented as `X \ (g X)`. There is also a special syntax for lists similar to the one used in standard prolog. "`,`" is used both as a list-element-delimiter and as a logical conjunction, "`;`" is a logical disjunction.

3. Basic ideas of the implementation

The implementation uses a dcg-like grammar with two additional arguments to record the list of constraints under which the current interpretation is valid.

For example:

```
sent (VP NP) C1 C3 L1 L3 :-
  np NP C1 C2 L1 L2,
  vp VP C2 C3 L2 L3.
```

says that the interpretation `(VP NP)` of the sentence represented by the d-list `L1-L3` is the interpretation of a verb-phrase `VP` applied to the interpretation of a noun-phrase `NP`, under the condition of the conjunction `C1-C3` of the constraints represented by the two d-lists `C1-C2` and `C2-C3`. The discharge rules can be applied if certain conditions on the types of the terms constituting the interpretation are met. In addition to this certain conditions on the occurrences of variables in the set of constraints must be met. The types of the interpretations are in general dependent on what rule produced the partial interpretation. This means that calls to the discharge-rules can be inserted in the structural rules of the grammar in appropriate places.

For example the grammar rule mentioned above actually has the form:

```
sent S C1 C4 L1 L3 :-
  np NP C2 C3 L1 L2,
  vp VP C3 C4 L2 L3,
  qprop (VP NP) S C2 C1 C4.
```

where qprop is a discharge-rule that may be applied to the the interpretation `(VP NP)` zero or more times, each time reducing the list of constraints `C1-C4` by one element, and producing a new interpretation in `S`. If sent is called with a goal of the form "`(sent S [] [] Sent [])`." the grammar will backtrack until it finds an interpretation of sent with an empty set of constraints.

So far no real use of the higher order unification procedure has been made. It is with the implementation of the discharge-rules that it becomes essential. The discharge-rules I have implemented have the following form in Pereira´s account.

$$\frac{\text{bind}(Q, x, R), \Gamma \vdash \text{p}\ \mathit{int}_{\iota \to o}\ T}{\Gamma \vdash \text{p}\ \mathit{int}_{\iota \to o}\ \lambda y.Q(R, \lambda x.T(y))}$$

$$\frac{\text{bind}(Q, x, R), \Gamma \vdash \text{p}\ \mathit{int}_{o}\ S}{\Gamma \vdash \text{p}\ \mathit{int}_{o}\ Q\ (R, \lambda x.S)}$$

These rules are accompanied by a set of constraints on the variables occurring in the sequents. For example, in none of these rules may x occur free in Γ.

The basic idea of the implementation is to use higher order unification to effect the bindings in the above expressions, and thereby get a grammar that produces unconditional interpretations from sentences in natural language.

4. Description of the implementation

This section contains a detailed account of the implementation. Many of the methods used are quite obvious, and are listed merely for the sake of completeness. The description of the implementation of the discharge-rules together with the discussion of the problems that appeared contain the main results.

4.1. Types

Lambda Prolog is a typed language. It contains a type-inference system that is very similar to that of ML. It is not necessary to specify any types at all, but doing so decreases the possibility that the unification mechanism will go into an infinite computation. The type specification constitutes an essential part of the program in another sense as well. In many cases we get more general answers if we do not specify the types than when we do, and this can be very valuable. As we will see this is sometimes not quite enough, and we get answers that are still more general than the ones we would expect. It is quite possible that a more expressive type-system would solve this problem.

The first set of type-declarations given below are the only ones necessary. All others are inferred by the system. I include some of these as examples.

```
type sent      % Type of the sentence grammar rule
      bool ->
      (list bool) -> (list bool) -> (list word) -> (list word) -> o.
type qpred     % Type of the qpred discharge rule
      (i -> bool) -> (i -> bool) ->
      (list bool) -> (list bool) -> (list bool) -> o.
type bind      % The type of a left-hand sequent element (a constraint)
      ((i -> bool) -> (i -> bool) -> bool) -> i -> (i -> bool) ->
      bool.
```

The following are inferred:

```
% type det ((i -> bool) -> (i -> bool) -> bool) ->      % A determiner
%       (list bool) -> (list bool) -> (list word) -> (list word) -> o.
% type noun (i -> bool) ->                              % A noun
%       (list bool) -> (list bool) -> (list word) -> (list word) -> o.
% type itv (i -> bool) ->                     % An intransitive verb
%       (list bool) -> (list bool) -> (list word) -> (list word) -> o.
% type tv (i -> i -> bool) ->                 % A transitive verb
%       (list bool) -> (list bool) -> (list word) -> (list word) -> o.
% type rnoun (i -> i -> bool) ->              % A relational noun
%       (list bool) -> (list bool) -> (list word) -> (list word) -> o.
% type qprop bool -> bool ->                  % The qprop discharge rule
%       (list bool) -> (list bool) -> (list bool) -> o.
```

4.2 Structural rules

The implementation of the structural rules are pretty straightforward. The result of applying the interpretation of a verb-phrase on the interpretation of a noun-phrase may discharge if the list of constraints is nonempty, and the variable-constraints for the discharge-rule are filled. This is implemented in the predicate qprop. Similarly for verb-phrases and common nouns with the predicate qpred.

```
sent S C1 C4 L1 L3 :-                              % A sentence
  np NP C2 C3 L1 L2,
  vp VP C3 C4 L2 L3,
  qprop (VP NP) S C2 C1 C4.                         % May discharge...

vp V C1 C2 L1 L2 :-                                % A verb-phrase
  itv V C1 C2 L1 L2.
vp VP C1 C4 L1 L3 :-
  tv Verb C2 C3 L1 L2,
  np NP C3 C4 L2 L3,
  qpred (Obj \ (Verb Obj NP)) VP C2 C1 C4.          % May discharge...

np X [(bind D X N) | C1] C3 L1 L3 :-               % A noun-phrase
  det D C1 C2 L1 L2,
  cn N C2 C3 L2 L3.

cn N C1 C2 L1 L2 :-                                % A common noun
  noun N C1 C2 L1 L2.
cn CN C1 C4 L1 L3 :-
  rnoun Noun C2 C3 L1 L2,
  np NP C3 C4 L2 L3,
  qpred (Obj \ (Noun Obj NP)) CN C2 C1 C4.          % May discharge...
```

4.3. Dictionary

The dictionary contains the terminal symbols of the grammar. Each entry gives the semantics of the given input word. The semantics of a word is in all cases a symbol representing a function of some type, whereas the input stream is a list of objects of type "word".

```
det exists C C [a | L] L.
det iota C C [the | L] L.
det forall C C [every | L] L.

noun dog_1 C C [dog | L] L.
noun child_1 C C [child | L] L.

rnoun friend_1 C C [friend, of | L] L.

itv sleeps_1 C C [sleeps | L] L.
itv comes_1 C C [comes | L] L.
```

4.4. The discharge rules

The discharge rule´s first argument is the representation of the input-semantics of the input-stream. The discharge rule will return a new semantics as its second argument, and a new list of constraints in accordance with Pereira´s theory.

If the output list is unconstrained, the discharge-rules will produce all interpretations, given all possible remaining lists of constraints, upon backtracking.

```
qpred T T C1 C1 C2.                          % Basis case
qpred (F X) T C1 C3 C4 :-                    % Discharge rule for predicates
  delete (bind Q X R) C1 C2 C4,
  independent X C2 C4,                       % Not implemented (see below)
  qpred (Obj \ (Q R (X \ ((F X) Obj)))) T C2 C3 C4.

qprop S S C1 C1 C2.                                 % Basis case
qprop (F X) S C1 C3 C4 :-                           % Discharge rule for
  delete (bind Q X R) C1 C2 C4,                     % sentences
  independent X C2 C4,                              % Not implemented
  qprop (Q R (X \ (F X))) S C2 C3 C4.
```

The definitition of `delete` would be trivial if Lambda prolog had an occurs-check. As it does not, the current implementation actually implements a simplified occurs-check on lists.

A problem is the `independent` predicate which is supposed to guarantee that the variable X does not occur in the remaining constraints `C2-C4`. The predicate independent currently does nothing. For a discussion of this see section 5. For some examples of how this unsoundness of the implementation effects the answers that are generated see section 6.

Note however how elegantly the higher order unification handles the lambda-binding of the free variable `X` occurring in `(F X)`, where `F` is unified with the (in a certain sense) simplest function of `X` unifiable with the input-semantics. The `X` occurring in the recursive call to the discharge rule is of course not the same variable `X` but a new lambda bound variable. In `qpred`, `(F X)` is applied to another new lambda-bound variable `Obj`.

5. Some problems.

A real problem (as Pereira also suggests in his paper) seems to be the handling of the variable constraints in the discharge rules. Some of the constraints are handled automatically by the higher order unification in a very natural way. But when it is not (the case where the variable which we are abstracting over may not occur in the rest of the constraints), there seems to be no simple way to express constraints of this kind even in the theoretically very powerful syntax of Lambda Prolog. We can not traverse the data structure representing the functions, as the higher order unification handles these as real functions and a recursion over such a structure is not guaranteed to terminate. The same problem would appear in an implementation of the capture rules, where the variable constraints are even more complex.

Pereira suggests that the work of Harper, Honsell and Plotkin [6] may give us a way to formulate a logic in which such variable constraints are not needed. This may be a way of doing in LF, but in my case, this would amount to implementing a subset of LF in lambda prolog, and in doing this I would encounter exactly the same types of problems as the ones I was trying to avoid. An other alternative would be to study alternative and more expressive type systems for incorporation into the logic programming framework. This may provide the expressiveness we obviously lack here.

We could also get around the problem with a quite dirty hack if Lambda Prolog had an occurs-check. Considering the complexity of the unification problem in these kind of theories, the cost of an occurs-check should not be prohibitive.

6. Examples.

6.1. The semantics of a simple sentence.

```
?- % ---- Enter: ----
sent Sem [] [] [the, dog, sleeps] [].

Sem = iota dog_1 X \ (sleeps_1 (Var50 X))

 The remaining flexible - flexible pairs:
<Var42 , Var50 Var42>
```

The desired result is actually the function: `(iota dog_1 (X \ (sleeps_1 X)))`. Instead we get the more complicated answer above under the the constraint that the function variable Var50 is unifiable with a function that returns its first argument as a value. This seemingly unnecessary generality is derived from Huet´s higher order unification procedure, and ultimately from the undecidibility of higher order unification. This comment applies to all the examples below as well.

6.1. The semantics of a common noun.

```
?- % ---- Enter: ----
cn Sem [] [] [friend, of, every, child] [].

Sem = Obj \ (forall child_1 X \ (friend_1 Obj (Var59 X Obj)))

 The remaining flexible - flexible pairs:
<Var63 \ (Var59 Var15 Var63) , Var95 \ Var15>
```

This is a more complicated constraint on the function variable `Var59`. It does seem intuitively correct however.

6.3. A more complex sentence with alternative scopings produced on backtracking.

```
?- % ---- Enter: ----
sent Sem [] [] [a, friend, of, every, child, comes] [].

Sem = forall child_1 X \ (exists Var142 \ (friend_1 Var142 (Var135 X
Var142)) W2 \ (comes_1 (Var140 X W2)))

 The remaining flexible - flexible pairs:
<Var139 \ (Var135 Var36 Var139) , Obj \ Var36>
<Var83 , Var140 Var36 Var83>
% ---- Enter: ----
;

Sem = exists Var283 \ (friend_1 Var283 Var36) X \ (forall Var304 \
(child_1 Var304) W2 \ (comes_1 (Var302 X W2)))

 The remaining flexible - flexible pairs:
<Var288 , Var302 Var288 Var36>
```

Note how the unsoundness of the implementation of the discharge rules here produces an incorrect result. The Variable Var36 is free in the above expression. This is exactly the case that the variable constraint is supposed to exclude.

```
% ---- Enter: ----
;

Sem = exists Obj \ (forall child_1 X \ (friend_1 Obj (Var455 X Obj)))
Var530 \ (comes_1 (Var529 Var530))

 The remaining flexible - flexible pairs:
<Var521 , Var529 Var521>
<Var459 \ (Var455 Var36 Var459) , Var507 \ Var36>
% ---- Enter: ----
;

 no
```

6 Conclusions

Although some important problems remain to be solved, the success so far of such a simple and straightforward implementation of Pereira´s ideas seems very promising. If we could either state the discharge-rules in a logic in which the variable constraints were simpler, or find a way to cleanly express the constraints in Lambda Prolog, we would have the basis of a very powerful system indeed. The interpretation produced is a clearcut representation of Montague Semantics, and the result is available to a reasoning system in a well-defined (though very complex) logical language. It would be very interesting to investigate how a somewhat more complete system would perform on the kind of problems where Montague Semantics really excels, e.g. epistemic and intentional constructs in natural language.

References

[1] Fernando C. N. Pereira. *Towards a Deductive Theory of Sentence Interpretation.* Proceedings of "Recent Developments and Applications of Natural Language Understanding" seminar in London December 87.

[2] Miller D. A. & Nadathur G. *Higher-Order Logic Programming.* Proceedings of the Third International Logic Programming Conference, Imperial Collage, London, England, July 1986.

[3] Huet. G. P. *A unification algorithm for typed λcalculus.* Theoretical computer science I (1975) 27-37.

[4] Nadathur G. *A Higher-Order Logic as the Basis for Logic Programming.* Ph.D. Dissertation, University of Pennsylvanina, May 87.

[5] Miller D. A. & Nadathur G. *Some uses of Higher-Order Logic in Computational Linguistics.* Proceedings of the 24th Annual meeting of the association for Computational Linguistics, 1986, 247 - 255.

[6] Harper R. & Honsell F. & Plotkin G. *A framework for defining Logics.* In Proceedings of the Second Symposium on Logic in Computer Science, Cornell University, IEEE, Ithaca, New York, 1987