Meta-Programming in Logic Programming

Meta-Programming in Logic Programming

edited by Harvey Abramson and M. H. Rogers

The MIT Press
Cambridge, Massachusetts
London, England

This book was printed and bound in the United States of America

Library of Congress Cataloging-in-Publication Data

Meta-programming in logic programming / edited by Harvey Abramson and M.H. Rogers.
p. cm. — (Logic Programming)
ISBN 0-262-51047-2
1. Logic programming. I. Abramson, Harvey. II. Rogers, M. H., 1930– . III. Title: Title: Meta-programming. IV. Series.
QA76.63.M47 1989
006.3—dc20 89-35120
CIP

Series Foreword

The logic programming approach to computing investigates the use of logic as a programming language and explores computational models based on controlled deduction.

The field of logic programming has seen a tremendous growth in the last several years, both in depth and in scope. This growth is reflected in the number of articles, journals, theses, books, workshops, and conferences devoted to the subject. The MIT Press series in logic programming was created to accommodate this development and to nurture it. It is dedicated to the publication of high-quality textbooks, monographs, collections, and proceedings in logic programming.

Ehud Shapiro
The Weizmann Institute of Science
Rehovot, Israel

Foreword

A meta-program is any program which treats another program as data. The language in which the meta-program is written is usually called the meta-language, and the language of the program which is the data for the meta-program is called the object language. This is a wide ranging definition in that it includes such "familiar" meta-programs as compilers, editors, simulators, debuggers, program transformers, and so on. When the meta-language and object-language are identical, it also includes "meta-circular interpreters", i.e., interpreters for a language which are written in the language being interpreted.

Meta-programming is a subject therefore of considerable practical and theoretical interest, and has been for some time. There is an added dimension to this interest when it becomes easy to write, test, and consider the implications of meta-programs. Most of the meta-programs mentioned above, where the meta-language and object language are different, are complicated objects, hard to write, hard to maintain, and hard to understand. Consider, however, the following well known "vanilla" interpreter for logic programs and pure Prolog:

$$solve(empty) \leftarrow$$
$$solve(x\&y) \leftarrow solve(x) \wedge solve(y)$$
$$solve(x) \leftarrow clause(x, y) \wedge solve(y)$$

Although it is possible to write meta-circular interpreters in other programming languages, LISP for example, they are not quite as concise as this one.

This standard vanilla interpreter leads into many practical and theoretical issues. On the practical side, to take a few examples, more flavors are wanted in the sense of being able to provide a complete definition of real programming languages such as Prolog (with some admittedly unsavoury features) and to implement sophisticated knowledge based systems, including expert systems. On the theoretical side, the simple vanilla interpreter leads into tricky questions of representation and of soundness and correctness of the interpreter.

In order to address these practical and theoretical problems, $\mathcal{META}_{88}$, a Workshop on Meta-Programming in Logic Programming was held at the University of Bristol, 22-24 June, 1988. This book is the result of that workshop, containing all but a few of the original papers presented there, and quite often, with the advantage of the more relaxed form of book publication, in an expanded and deepened form. [1] We shall let the papers speak for themselves, the collection as a whole representing a fairly comprehensive view of what meta-programming is about within the discipline of logic programming.

[1] The papers which do not appear here have been published elsewhere. Alan Bundy's The Use of Explicit Plans to Guide Inductive Proofs appears in Proceedings of CADE9, edited by Luck, R. and Overbeek, R., Springer-Verlag, and Harvey Abramson's Metarules and an Approach to Conjunction in Definite Clause Translation Grammars, appears in Logic Programming: Proceedings of the Fifth International Conference and Symposium, edited by Robert K Kowalski and Kenneth A. Bowen, MIT Press.

The $\mathcal{META}_{88}$ Workshop was organized by Prof. J.W. Lloyd who also edited the original, provisional Proceedings. The members of the Program Committee were

- Harvey Abramson, University of Bristol
- P.M. Hill, University of Bristol
- J.W. Lloyd, University of Bristol
- S.G. Owen, Hewlett-Packard Research Labs, Bristol
- M.H. Rogers, University of Bristol
- J.C. Shepherdson, University of Bristol

Many thanks go to Peter Phillips, LaTeX hacker extraordinaire, for his efforts in bringing a host of varied papers to a uniform style.

Harvey Abramson
M.H. Rogers
April 1989

Contents

Chapter 1

A Meta-Rule Treatment for English Wh-Constructions

Lynette Hirschman[1]

Paoli Research Center
Unisys Defense Systems

Abstract

This paper describes a general meta-rule treatment of English wh-constructions (relative clauses, and questions) in the context of a broad-coverage logic grammar that also includes an extensive meta-rule treatment of co-ordinate conjunction. Wh-constructions pose difficulties for parsing, due to their introduction of a dependency between the wh-word (e.g., *which*) and a corresponding *gap* in the following clause: *This is the book which I thought you told me to refer to ( )*. The gap can be arbitrarily far away from the wh-word, but it *must* occur within the clause, or the sentence is not well-formed, as in **The book which I read it*.

A meta-rule treatment has several advantages over an Extraposition Grammar-style treatment: a natural delimitation of the gap scope, the ability to translate/compile the grammar rules, and ease of integration with conjunction. Wh-constructions are handled by annotating those grammar rules that license a gap or realize a gap. These annotations are converted, via the meta-rule component, into parameterized rules. A set of paired input/output parameters pass the need for a gap from parent to child and left sibling to right sibling until the gap is realized; once the gap is realized, the parameter takes on a *no_gap* value, preventing further gaps from being realized. This 'change of state' in the paired parameters ensures that each gap is filled exactly once. The conjunction meta-rule operates on the parameterized wh-rules to link gaps within conjoined structures by unification, so that any gap within a conjoined structure is treated identically for all conjuncts.

[1]This work has been supported in part by DARPA under contract N00014-85-C-0012, administered by the Office of Naval Research; and in part by internal Unisys funding.

1.1 Introduction

Wh-constructions are one of the classically difficult parsing problems, because a correct treatment requires interaction of non-adjacent constituents, namely the wh-word, which introduces a constituent in clause-initial position, and the following construction which is missing a constituent (the *gap*). The gap can be arbitrarily far from the introducing wh-word (an *unbounded dependency*); in particular, it can appear within deeply embedded constructions, such as *the person that [I had hoped [Jane would tell [( ) to get the books]]]*, where there are three levels of embedded structure. It is possible, in principle, to write a rule for each case where a gap can appear. However, since the number of constructions which can accommodate a gap is very large (e.g., most complement types), this is both extremely labor-intensive and unmaintainable from the grammar writer's point of view.

It is also possible to write general rules for gap-realization, e.g., a noun phrase can be realized as a gap. If this approach is taken, then these rules must be carefully constrained to accept gaps only when inside a wh-construction; in addition, the wh-construction must contain exactly *one* gap. These restrictions involve complex and expensive search up and down the parse tree, to determine whether a gap is occurring inside a wh-construction.

In many ways, the wh-problem parallels the problem of co-ordinate conjunction that has also been a major obstacle for natural language systems. Both constructions involve gaps, both affect large portions of the grammar, and both require a major modification to the grammar and/or to the parsing mechanism to handle the linguistic phenomena.

There have been two basic approaches to conjunction and wh-constructions in the computational linguistics literature: modification of the parser (interpreter) and meta-rules. Of these, the first approach has been far more common. For conjunction, a number of variants on the 'interrupt' driven approach have been presented, both in conventional natural language processing systems [13, 12, 14], and in the context of logic grammars [4]. The same is true for logic grammar implementations of wh-constructions: the most generally used treatment is the interpreter-based treatment of Extraposition Grammar (XG) [10].

Meta-rules offer an appealing alternative to interpreter-based approaches, both for conjunction and for wh-expressions. Meta-rules are particularly well-suited to phenomena that range over a variety of syntactic structures, where the linguistic description would otherwise require regular changes to a large set of grammar rules. The use of meta-rules turns out to be efficient computationally. It also preserves compactness of the underlying grammar, so that the grammar is still maintainable from the point of view of the grammar-writer. Finally, the meta-rule approach avoids additional interpretive overhead and permits translation/compilation of grammar rules for efficient execution [5].

For conjunction, the meta-rule approach forms the basis for a comprehensive

treatment of co-ordinate conjunction in Restriction Grammar [7]. Abramson has provided a generalization of this approach, formulating meta-rules as a specialized case of meta-programming [2]. Other researchers have also examined a meta-rule approach to related phenomena; Banks and Rayner, for example, have proposed a meta-treatment of the comparative [3].

For wh-constructions, we propose here an approach based on parameterization of the grammar rules. This is similar in spirit to the GPSG notion of 'slash categories' [6], but in the framework of logic grammar. The use of parameterized rules to pass gap information has previously been proposed in a logic grammar framework, specifically as *gap-threading* [10, 11]. Our approach differs from Pereira's in several ways, the most important of which is the use of meta-rules. The meta-rule approach provides a much cleaner user interface, making it possible for the grammar writer to use linguistically motivated annotations to indicate gap license and gap realization for the unparameterized BNF definitions in the grammar. The meta-rules process these annotations to generate parameterized grammar rules which, in turn, can be translated and compiled for efficient execution. The meta-rule treatment also has the property of combining seamlessly with a meta-rule treatment of co-ordinate conjunction.

1.2 Wh-Constructions: The Linguistic Issues

Wh-constructions are one instance of a class of problems referred to as *unbounded dependencies* – that is, constructions where the interdependent entities may be arbitrarily far apart. In the case of wh-constructions, we have a wh-expression which begins the clause (e.g., *who, what, which, whose book, how*, etc.) followed by a *gap* at some later point in the clause. The wh-expression may take the place of a noun phrase, an adjective phrase or an adverbial phrase. These may appear in the subject, object or sentence adjunct positions.

As the sentences of Figure 1.1 illustrate, there are a variety of wh-constructions, namely, relative clauses (including the zero-complementizer case, where an overt wh-word is absent, as in *the person I saw*), indirect questions (*I don't know what they mean*), wh-questions (*What do you want?*), and headless relatives (*You get what you deserve*). In addition to these basic types of wh-construction, there are also some constructions where the wh-expression is embedded inside a noun phrase (*this is the person whose mother I met*), with the wh-word *whose* modifying a noun phrase; the subsequent gap is filled by the noun phrase (*the person's mother*) of which the wh-word is a part. There are also wh-constructions embedded in prepositional phrases, as in *the person from whom I learned it* or *the door the key to which is missing.*

A wh-construction involves (1) a *wh-word* (e.g., *who*) contained in a clause-initial *wh-expression*; and (2) a *gap*: a constituent omitted in the clause following the wh-word, e.g., *the book which I bought ( )*. Relative clauses also have an antecedent for the relative pronoun (the wh-word); for questions, the wh-word

marks the questioned item.

Wh = who; gap = subject NP
The person who () was here
Wh = who(m); gap = object NP
Who did you see ()?
Wh = that; gap = object NP
The time that I spent ()
Wh = that; gap = sentence adjunct adverbial
The time I visited them ()
Wh = who(m); gap = embedded object
The person who they told me they had tried to visit ()
Wh = who; gap = embedded subject
Who did they tell you () had visited them?
Wh = how; gap = sentence adjunct adverbial
Do you know how they did it ()?

Figure 1.1: Wh-constructions in English

To regularize a wh-construction, the wh-expression fills in the gap, and the wh-word is replaced by its antecedent (if in a relative clause).[1] For example, in the phrase *the movie which I saw ( )*, the wh-expression is *which* and the gap is after *saw*. Moving the wh-expression into the gap, we get *the movie [I saw which]*. Then, replacing *which* by its antecedent (*the movie*), we get: *the movie [I saw the movie]*. Similarly for questions, we get *what did you see ( )?* regularized as *did you see what?*. In some cases, however, the wh-word is not identical to the whole wh-expression, as in *the bird whose nest I found ( )*. Here, the wh-expression is *whose nest*, and the wh-word is *whose*. Again, we replace the gap (the object of *found*) by the wh-expression, to get *the bird [I found whose nest]*. Then we replace the wh-word by its antecedent, namely *the bird*: *the bird [I found the bird's nest]*, preserving the possessive marker from whose. Similarly in a question, we get: *which book did you read ( )* regularized as *did you read which book?*. To summarize, wh-expressions are introduced by a phrase containing a wh-word; following a wh-expression, there must be a gap, and this gap is understood as the wh-expression, after it has had the antecedent of the wh-word word filled in (if in a relative clause).

[1]The expression *replace by its antecedent* is used loosely here. What is really meant is replacing the relative pronoun by a pointer to the antecedent. This preserves co-referentiality of the relative pronoun and its antecedent, and avoids the dangers of copying quantifier and other modifier information.

The need for a gap can be captured very simply by associating with each grammar definition a set of paired input/output parameters. The input parameter signals whether or not a gap is need when the node is about to be constructed, at rule invocation time. The output parameter signals whether that need has been satisfied once the node is completed, at rule exit. Thus an assertion in a relative clause has as its input parameter the need for a gap (**need_gap**) and on exit, that need must have been satisfied (indicated by a **no_gap** output parameter). These parameters, once set, are simply passed along from parent to child, and sibling to sibling, via unification through linked input/output parameters.

However, an assertion may also occur as the main clause, where it is not licensed for a gap. This is illustrated in Figure 1.2 by the (simplified) definition for a sentence, as having two alternatives: an assertion or a question. The definition for *assertion* itself therefore must be neutral with respect to gaps, since that depends on where it is called from (relative clause or sentence). The parameters in the assertion definition simply pass along the information from parent to child and sibling to sibling. If the assertion is in a relative clause, then the need for a gap is passed along until some node (*nullwh* in Figure 1.2) realizes the gap (that is, accepts the empty string), at which point its output parameter is set to **no_gap**; this is passed along and finally, back up to assertion. If the assertion occurs as the main clause of a sentence, it has no need for a gap and in fact, cannot unify with the gap realization rule, which requires an input parameter of **need_gap**.

This mechanism enforces the constraint that only a node with the parameter pair (**need_gap/no_gap**) can dominate a gap. Any node whose input and output parameters are equal has not 'changed state' – that is, whatever it needed (or didn't need) on rule entry, it will still need at rule exit. Procedurally, any rule whose input and output parameters are equal cannot unify with the gap realization rule. The flow of information through the tree is illustrated in Figure 1.3.

1.3 The Framework: Restriction Grammar

The proposed solution is presented in the context of Restriction Grammar [8], which is the syntactic portion of the PUNDIT text processing system [9]. However, this solution is only dependent on a few general properties of Restriction Grammar, which it shares with other formalisms (e.g., Definite Clause Translation Grammars [1]). A Restriction Grammar is written in terms of context-free BNF definitions, augmented with constraints (*restrictions*) on the well-formedness of the resulting derivation tree. Constraints operate on the derivation tree, which is constructed automatically during parsing; restrictions traverse and examine this tree, to determine well-formedness.

One of the significant characteristics of Restriction Grammar is the *absence* of parameters. Context sensitivity is enforced by the restrictions, which obtain information from the derivation (parse) tree, rather than via parameter passing.

```
% Simplified BNF definitions before parameterization for
 wh-constructions:

sentence     ::= assertion; question.
rel_clause   ::= wh, assertion.
assertion    ::= subject, verb, object.
subject      ::= noun_phrase.
verb         ::= *v.                % * indicates terminal lexical
 category
object       ::= noun_phrase; assertion;....

noun_phrase ::= lnr; *pro.
noun_phrase ::= nullwh.
lnr          ::= ln, *n, rn.       % noun with left, right adjuncts.
rn           ::= null; pp; rel_clause.

null         ::= % .       % empty string (for empty adjunct slots)
nullwh       ::= % .       % empty string for gap realization.

wh           ::= [who]; [which]; ....

% Parameterized BNF definitions for handling relative clause:
% Where parameters pass no information, input = output parameter.

sentence(X/X)       ::= assertion(no_gap/no_gap); question
(need_gap/no_gap).
rel_clause(X/X)     ::= wh(Y/Y), assertion(need_gap/no_gap).
assertion(In/Out)   ::= subject(In/Subj), verb(Subj/Verb), object
(Verb/Out).
subject(In/Out)     ::= noun_phrase(In/Out).
verb(In/In)         ::= *v.
object(In/Out)      ::= noun_phrase(In/Out); assertion(In/Out); ...

noun_phrase(In/In) ::= lnr(In/In); *pro.
noun_phrase(need_gap/no_gap)
                    ::= *nullwh.                    % empty string
                                                    %      for gap
lnr(In/In)          ::= ln(In/In), *n, rn(In/In).   % noun + left,
                                                    % right adjuncts
rn(In/In)           ::= null(In/In); pp(In/In);
                          relative_clause(In/In).

null(In/In)         ::= % . % empty string (for empty adjunct slots)
nullwh(need_gap/no_gap)
                    ::= % . % empty string for gap realization.
wh(In/In) ::= [who]; [which]; ....
```

Figure 1.2: Simplified Rules with Parameters for Wh.

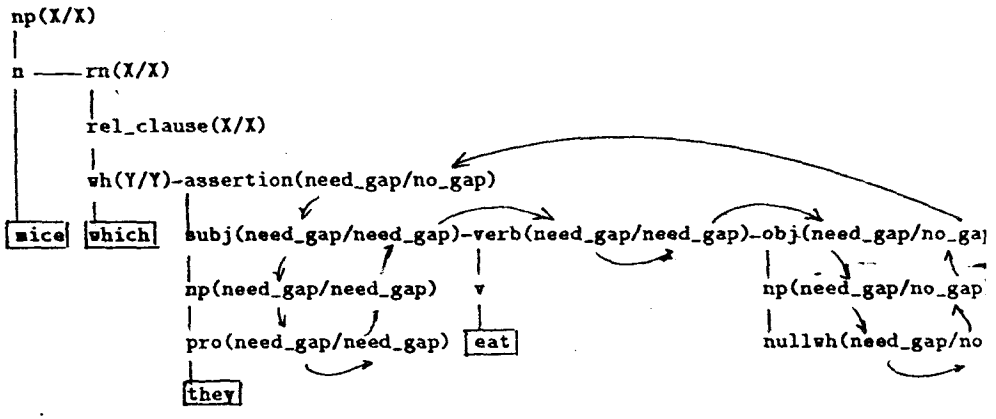

Figure 1.3: Flow of Information in *...mice which they eat.*

Restriction Grammar is implemented as a form of logic grammar which includes parameters not just for the word stream, as in DCG's, but also for the automatic construction of the derivation tree as well. In addition, each grammar rule is augmented with an associated regularization rule (indicated by a right hand arrow $\rightarrow$), which incrementally constructs an *Intermediate Syntactic Representation (ISR).* The ISR is an operator/operand notation that represents a canonical, regularized form of the parse tree. The regularization rule composes the ISRs of the daughter nodes in the derivation tree into the ISR of the current node, using lambda reduction. Computation of the ISR for wh-constructions is discussed in greater detail in section 6.

1.4 The Solution: Meta-Rules

Although parameterization is an elegant and efficient solution, it presents a major problem – it obscures the declarative aspect of the BNF rules, and correct parameterization of rules can be tedious and error prone, especially since there are some 40 object types in our current broad-coverage grammar of English.

The solution is to define a set of annotations to express the required linguistic constraints: gap introduction via wh-word, gap realization, and gap prohibition. Figure 1.4 shows a grammar using annotations defined as prefix operators applied to the node names in BNF definitions. Gap introduction is written as <<, gap realization as >>, and gap prohibition as <>. These are used, for example, to flag the need for a wh-word in a wh-expression, followed by the need to realize a

gap:

```
rel_clause ::= <<wh, >>assertion.
```

Annotations can appear on either the left-hand side or the right-hand side of BNF definitions. By introducing the gap-requirement on the right-hand side of a BNF definition, we create a conditional gap requirement. For example, *assertion* requires a gap in the context of a relative clause, but not as the normal realization of a sentence (main clause) option. Thus we do not want to annotate the definition for *assertion*, but the call to *assertion* in *rel_clause*. However, the definition for *nullwh* is always a gap realization rule, hence it is annotated on the left-hand side (see Figure 1.4).

In certain cases, we need to define a special gap-requirement rule. For example, we define a special case for noun-phrase gap realization. This enables us to *block* transmission of gap parameters in all other options of noun phrase. To do this, we use the third annotation <> to set input parameter equal to output parameters. This annotation is also used to show that the verb can never license a gap. Similarly, the determiner (*det*) and pre-nominal adjective (*adjs*) rules cannot license a gap.

The remainder of the rules require no annotation; their parameters simply transmit whatever gap information is passed in. Figure 1.5 shows the parameterized definitions corresponding to the annotated definitions used in Figure 1.4.

1.5 The Meta-Rule Component

The meta-rule component for parameterization is implemented as a general procedure which adds parameters to each production in the grammar. At grammar read-in time, each rule is parsed and parameterized appropriately, depending on its annotation. The basic case is no annotation, in which case the following rules apply (*Label* is the left-hand side of the BNF definition; *Rule* is the right-hand side):

```
% Basic case:
wh_params(Label,Rule,NewLabel,NewRule) :-
    check_head_params(Label,InParam/OutParam,NewLabel),
    take_apart(Rule,NewRule,InParam,OutParam),!.

check_head_params(Label,Params,NewHead) :-
        insert_param(Head,Params,NewHead).
insert_param(Head,Params,NewHead) :-
        NewHead =..[Head,Params].

% Conjunction
take_apart((A,B),(NewA,NewB),InParam,OutParam) :- !,
```