Declarative semantics of a meta-programming language*

Henning Christiansen

Roskilde University, Computer Science Dept.,
P.O.Box 260, DK-4000 Roskilde, Denmark
E-mail: henning@ruc.dk

Abstract. We present a logical programming language, generative clause programs, which generalize definite clause programs with a meta-programming level. The fundamental notion in the language is that of a meta-goal which consists of a plain goal together with a program in which it is expected to be true. The declarative semantics can be defined in terms of Herbrand models consisting of such meta-goals. The well-known fixpoint characterization of the least Herbrand model for definite clause programs generalizes immediately. The procedural semantics is considered in terms of an interpreter program and a compiler for the language.

1 Introduction

A meta-level in logic programming is traditionally introduced by means of a self-interpreter programmed in the logical language itself, e.g., (Bowen, Kowalski, 1982, Bowen, Weinberg, 1985, Bowen, 1985, Hill, Lloyd, 1988, Subrahmanian, 1988). In this way, the access to the meta-level is syntactically just a call of a predicate and consequently, its semantics must be understood in terms of the interpreters internal mode of operations. In order to achieve a declarative understanding of meta-programming we need a view of the language in which the fundamental syntactic and semantic categories reflect the existence of the meta-level. We propose a language of *generative clause programs* which in general is concerned with *meta-goals*. A meta-goal is a goal together with an expression of the program in which it is expected to be true. The semantics of this language can be described in terms of Herbrand models consisting of meta-goals. The fixpoint characterization known from ordinary logic programs (van Emden, Kowalski, 1976) of the least such model generalizes immediately. The semantic definition covers embedded meta-levels in a natural way.

Overview

Section 2 introduces the syntax for generative clause programs and compares it with MetaProlog (Bowen, Weinberg, 1985, Bowen, 1985). As in MetaProlog we apply a ground representation for program text in order to avoid a confusion between variables

* Proceedings of META90, Workshop on Meta-Programming in Logic, (M. Bruynooghe, ed.). 4-6 April 1990, Leuven, Belgium, pp. 150-168, 1990.

at different levels. However, instead of using universal quantifiers to "declare" variables, we use a more compact notation so that the relation between a text and the term it denotes can be specified by a simple, compositional mapping.

An intuitive semantic definition is given in section 3. Section 4 provides a more precise explanation based on a fixpoint characterization of the least Herbrand model for generative clause programs.

In section 5 we illustrate the procedural semantics by an interpreter for our language programmed in Prolog. However, we can only execute meta-goals with fully given program parts. A full understanding of the procedural semantics in terms of a proper generalization of SLD-resolution (cf. Lloyd, 1987) which covers the entire formalisms is still lacking. Section 6 is a sketch of our compiled implementation of generative clause programs. A summary is given in section 7.

2 A logical language for meta-logical programming

The language of *generative clause programs* generalizes definite clause — or pure Prolog — programs, from which we take the basic syntactic notions.

Definition 1. The notions of *constants*, *variables*, *functors*, *terms*, and *ground terms* are defined as usual.
□

Programs are described by ground terms denoting them. However, the programs denoted by these terms contain, of course, usual logical variables.

A logical goal represents a statement which is expected to be provable in some program. In a meta-programming environment there are many simultaneous programs involved and therefore we introduce the notion of a meta-goal which consists of a goal together with the program in which it is expected to be provable. The symbol "in" is a particular binary functor.

Definition 2.A *meta-goal* is a term of the form

G in P.

The subterms G and P are called *goal* and *program parts*, respectively.

A *generative clause* is a term of the form

M_0:– $M_1, \ldots, M_n, \quad n \geq 0,$

where $M_0, \ldots, M_n$ are meta-goals.

A *generative clause program* is a list of generative clauses.
□

The intended meaning is that program parts instantiate to terms that represent programs. On the left hand side, the program part provides a handle to the "current" program in which the instance of a rule is applied; program parts on the right hand side resemble the program argument to calls of an interpreter. Hence, with respect to expressibility, generative clause programs are equivalent to MetaProlog (Bowen, Weinberg, 1985, Bowen,

1985) — the difference is at the conceptual level: Our language is concerned with meta-goals whereas MetaProlog is concerned with ordinary goals, some of which happen to be calls of an interpreter program.

The fact that we view a program as a list of clauses, as opposed to a set, emphasizes the representability of programs within themselves.

Note. Program parts in the head of a rule may in principle express strange conditions on the applicability of the rule. Writing, e.g., [–, –, –] would disable a rule in all but three-rule programs. We believe that in any reasonable program, this program part will consist of a variable. However, for simplicity we will not put any syntactic restrictions into the language for this purpose.
□

Our notation for (terms denoting) variables is inspired by Colmerauer's (1975) original notation, however here generalized for meta-level programming. A variable is denoted by a constant preceded by an asterisk. A symbol intended to serve as a variable in a generated program — but considered a ground term at the base level — is preceded by two asterisks, and so forth for further embedded meta-levels.

The relation between certain ground terms and the programs they denote is defined by a function, $\mathcal{D}$, below. We assume the existence of a bijective function, $\mathcal{V}$, from constants to variables.

Definition 3. The *syntactic denotation function* for generative clause programs is the partial function, $\mathcal{D}$, from the set of ground terms to the set of terms defined inductively as follows.

$\mathcal{D}[\![c]\!] = c$, for any constant, c,

$\mathcal{D}[\![*c]\!] = \mathcal{V}[\![c]\!]$, for any constant, c,

$\mathcal{D}[\![*^n c]\!] = *^{n-1}c$, for any constant, c, and $n > 1$,

$\mathcal{D}[\![f(t_1, \ldots, t_n)]\!] = f(\mathcal{D}[\![t_1]\!], \ldots, \mathcal{D}[\![t_n]\!])$, f different from unary *,

for any other term, t, $\mathcal{D}[\![t]\!]$ is undefined.

Whenever $\mathcal{D}[\![t]\!] = t'$, we say that *t denotes t′*.
□

Observation. Any term which includes a variable denotes nothing. Similarly for a term with a misplaced asterisk such as in *f(a). For any term without such misplaced asterisks, there exists a ground term denoting it. I.e., all sensible terms are denotable.
□

For example, the term *x denotes a variable whereas **x denotes a term which denotes a variable. The term,

```
p(*x, *y) in *prog:–
    q(*x) in [(q(**x) in **prog:– *y) | *prog]
```

denotes the following clause; the capital letters indicate "real" variables.

p(X, Y) in P:–
    q(X) in [(q(*x) in *prog:– Y) | P]

The meta-goal in the body contains a subterm, which serves as a template for a new rule to extend the current program. If Y is instantiated to, say, "r(*x, **z) in *prog", the template will denote the following rule.

q(X) in P:– r(X, *z) in P

Notational shorthands. The goals in a generative clause need not be equipped with explicit program parts. This is not part of the formal definition but is viewed as a notational convention. For any generative clause, the following shorthands apply in the given order:

- If, in a meta-goal, $M_i = G_i$ in P_i, in the body of a clause, the program part, P_i, is identical to the program part for the head of the clause, M_i can be written simply as G_i.
- If the program part in the head of a clause, $M_0 = G_0$ in P_0, is a variable which does not appear anywhere else in the clause, M_0 can be written simply as G_0.

□

This gives a natural injection of definite clause programs into the set of generative clause programs. The following rule, for example,

a:– b, c

is, according to our notational conventions, an abbreviated form of a rule denoted as follows.

a in *p:– b in *p, c in *p.

Our implementation of generative clause programs includes a *where*-notation and a function symbol, &, for combination of grammars, which allow the sample clause above to be written as follows.

p(*x, *y) in *prog:–
    q(*x) in *new_prog
where
    *new_prog = [*new_rule] & *prog,
    *new_rule = (q(**x) in **prog:– *y).

Compared to MetaProlog, the generative clause notation is marked by the following characteristics.

- There is no need for universal quantifiers ("all") in order to distinguish between variables at different levels.
- The level (base, meta, meta-meta, etc.) at which a given symbol is to serve as a variable is determined independently of the context in which it appears. Consider, e.g., the sample p(*x, **y, ***z).
- There is a clear structural resemblance between a term and whatever it denotes.
- There is no need for a catalogue of pseudo-predicates such as "current_theory(–)".
- The meta-programming facilities appear as an integral part of the language.

The latter property makes it more obvious to consider a declarative semantics.

3 Intuitive semantics

Plain definite clause programs are distinguished by their lucid intuitive and declarative semantics. The set of all ground goals provable according to a program can be described in a simple inductive fashion. Here we show a similar semantic characterization of generative clause programs. In section 4, we will give the precise details; here we consider a formulation in a style suited for elementary textbooks. (Compare with, e.g., (Bratko, 1986, p. 42)).

For comparison, we start by giving the definition for definite clause programs assuming that programs are given as terms denoting them.

Definition 4. The *provability relation for definite clause programs*, $\vdash_{\mathrm{DCP}}$, is defined inductively as follows.

- Whenever *pt* is a ground term which denotes a definite clause program, $\mathcal{D}[\![pt]\!]$, which contains a clause which has a ground instance

  $$g_0 \mathop{:\!-} g_1, \ldots, g_n$$

  such that

  $$\mathcal{D}[\![pt]\!] \vdash_{\mathrm{DCP}} g_i, \quad 1 \leq i \leq n,$$

  then

  $$\mathcal{D}[\![pt]\!] \vdash_{\mathrm{DCP}} g_0.$$

- $\vdash_{\mathrm{DCP}}$ holds for no other programs and goals.

□

This semantic definition is useful since:

- It is intuitively simple and requires only an elementary mathematical insight.
- It is independent of any implementation method.

These qualities are preserved when the definition is generalized to generative clause programs. The only difference is in the selection of the clause to be applied in a proof step.

Definition 5. The *provability relation for generative clause programs*, $\vdash_{\text{GCP}}$, is defined inductively as follows.

– Whenever pt_0 is a ground term which denotes a generative clause program, $\mathcal{D}[\![pt_0]\!]$, which contains a clause which has a ground instance

  $$g_0 \text{ in } pt_0\text{:-}\; g_1 \text{ in } pt_1, \ldots, g_n \text{ in } pt_n$$

  such that

  $$\mathcal{D}[\![pt_i]\!] \vdash_{\text{GCP}} g_i, \quad 1 \leq i \leq n,$$

  then

  $$\mathcal{D}[\![pt_0]\!] \vdash_{\text{GCP}} g_0.$$

– $\vdash_{\text{GCP}}$ holds for no other programs and goals.

□

Note. It should be made clear that the sort of clause instance referred to in this definition does *not* contain, in pt_0, a structure isomorphic to itself. What it contains is a representation of a clause of which it is an instance — and that is something quite different. The following is an example of such a clause instance.

p in [(p in *x:– q in *x)]:– q in [(p in *x:– q in *x)]

□

Observation. Let pt_{DCP} denote a definite clause program and pt_{GCP} be the text denoting its injection as a generative clause program. (The injection is given by our notational conventions described above). Then, for all ground goal instances, g,

$$\mathcal{D}[\![pt_{\text{DCP}}]\!] \vdash_{\text{DCP}} g$$

if and only if

$$\mathcal{D}[\![pt_{\text{GCP}}]\!] \vdash_{\text{GCP}} g.$$

I.e., DCP constitutes a proper sublanguage of GCP — with respect to the semantic as well as the syntactic aspects.
□

In the following we will study the precise meaning of the intuitive semantic definition.

4 A fixpoint characterization of the least Herbrand model

Here we define the declarative semantics of generative clause programs as a least Herbrand model. As for definite clause programs, this model can be characterized as the least fixpoint of a mapping from Herbrand interpretations to Herbrand interpretations (van Emden, Kowalski, 1976). Our development is quite traditional except that our Herbrand interpretations are concerned with meta-goals instead of with goals and hence we cover all possible programs by one and the same model. In the following, we adopt the concepts and notation of (Lloyd, 1987).

The following notion of self-containedness is central for the semantics of generative clause programs.

Definition 6. Let *pt* be a term which denotes a generative clause program, $\mathcal{D}[\![pt]\!]$. The set of all ground instances of clauses in $\mathcal{D}[\![pt]\!]$ is denoted $\mathcal{G}[\![pt]\!]$.

A ground instance of a generative clause,

$$C = \quad M{:}\!-M_1, \ldots, M_n$$

is said to be *self-contained* if

$$C \in \mathcal{G}[\![pt]\!], \quad \text{where}$$

$$M = g \text{ in } pt.$$

□

With respect to the soundness of definition 6, we refer to the note in the previous section.

For a definite clause program any instance of a rule can be applied freely in a proof, but in the generative case, we are only interested in those instances that are self-contained. In order to define a suitable notion of Herbrand models for generative programs, there are two possibilities. Either we can define the truth of a generative clause involving this constraint or we can develop the theory for a set of axioms consisting of all self-contained clause instances. For technical reasons we take the latter of these two approaches which otherwise are equivalent.

Definition 7. The set of *axioms of generative clause programs*, $\textsc{Axioms}_{\textsc{gcp}}$, is the set of all self-contained generative clause instances.

An *Herbrand interpretation for generative clause programs* is a set of ground meta-goals.

An *Herbrand model for generative clause programs* is an Herbrand interpretation, $\mathcal{M}$, such that for each

$$M{:}\!-M_1, \ldots, M_n \quad \in \textsc{Axioms}_{\textsc{gcp}},$$

it holds that

$$\{M_1, \ldots, M_n\} \subseteq \mathcal{M} \text{ implies } M \in \mathcal{M}.$$

□

$\text{AXIOMS}_{\text{GCP}}$ is equivalent to an infinite,[1] definite clause program concerned with a predicate "in" and thus we know from (Lloyd, 87) that the intersection of a non-empty set of Herbrand models itself is an Herbrand model. The intersection of all Herbrand models for generative clause programs is called the *least* Herbrand model and will be denoted $\mathcal{M}^{\text{GCP}}$.

Definition 8. The mapping T^{GCP} from Herbrand interpretations to Herbrand interpretations is defined as follows.

$$\begin{aligned}\text{T}^{\text{GCP}}(\text{I}) = \{M \mid &\text{ There exists}\\ &\quad M{:}\!-M_1, \ldots, M_n \quad \in \text{AXIOMS}_{\text{GCP}}\\ &\text{ such that}\\ &\quad \{M_1, \ldots, M_n\} \subseteq \text{I}\}\end{aligned}$$

□

T^{GCP} is identical to the usual transformation, T_{P}, defined for a definite clause program, P, where in this case P is identified with $\text{AXIOMS}_{\text{GCP}}$ as above. Hence we know from (Lloyd, 1987, chapters 1 and 2) that T^{GCP} is continuous and has a least fixpoint, $\text{lfp}(\text{T}^{\text{GCP}})$, and finally:

Theorem.(Fixpoint characterization of the least Herbrand model for generative clause programs).

$$\mathcal{M}^{\text{GCP}} = \text{lfp}(\text{T}^{\text{GCP}}) = \text{T}^{\text{GCP}}\uparrow\omega$$

□

The precise meaning of the intuitive semantic definition given in the previous section can now be stated as follows.

$$\{pt \text{ in } g \mid \mathcal{D}[\![pt]\!] \vdash_{\text{GCP}} g\} = \mathcal{M}^{\text{GCP}} = \text{lfp}(\text{T}^{\text{GCP}}) = \text{T}^{\text{GCP}}\uparrow\omega$$

5 An interpreter for generative clause programs

As a first step towards a procedural semantics for generative clause programs, we show a simple interpreter programmed in Prolog. However, it is only a partially correct implementation. The interpreter can only handle queries with fully given program parts. The declarative semantics above describes a language which in principle can be used to complete partially known programs such that a given goal can be proved.

The predicate "denotes" below maps a ground term to the term it denotes, however, with each variable consistently replaced by a completely new variable. This predicate is based heavily on the non-logical parts of Prolog; in case of a non-ground program part it stops with an error message.

[1] A definite clause program is usually defined to be a *finite* set of rules. However, it can be checked that the results referenced in this section also hold for infinite programs.

demo(G in Prog):–
member(FactTerm, Prog),
FactTerm denotes (G in Prog).

demo(G in Prog):–
member(RuleTerm, Prog),
RuleTerm denotes (G in Prog:– Body),
demo(Body).

demo((A, B)):– demo(A), demo(B).

The interpreter is called with a meta-goal whose goal part may contain usual Prolog variables whereas the program part is given in its ground representation.

The interpreter is very inefficient and serves only to illustrate the procedural aspects of the language. A generalization of the interpreter — or a similar generalization of SLD-resolution — to cover also incomplete program parts is still lacking.

6 Compiled implementation

The view of meta-level programming taken here invites to compiled implementations. In this section, we describe informally our implemented compiler which translates generative clause programs into Prolog code. As for our interpreter, only meta-goals with ground program parts can be executed this way. The implementation is based on dynamic compilation of dynamically generated clauses and a structure-sharing representation of programs that are variations of each other.

In the compiled approach, program parts of meta-goals will be instantiated to pointers to compiled program representations instead of the program source texts themselves. A first suggestion is to identify each particular program by a ground index, e.g., an integer number. New indices are created whenever a new program appears and is compiled into the common program space. A meta-goal is represented as a goal with the program index as an additional argument. So the following rule,

p(*x) in *prog:– q(*x, *y) in *prog, r(*y) in ⟨*new program*⟩,

belonging to a program with index *i* can be compiled into the following Prolog rule.

p(*i*, X):– q(*i*, X, Y), compile(⟨*new program*⟩, NewI), r(NewI, Y).

However, programs are often variations of other programs, e.g., one program being a slight extension of an already compiled program. In such cases we should apply a structure sharing representation such that each rule is compiled only once. A set of programs thus embedded in each other determines a partial ordering. If $program_j$ is an extension of program $program_i$, we say that:

$program_i < program_j$

We now choose a set of program indices which respects this ordering, namely open-ended lists ordered according to their degree of instantiation, e.g.,

[0, 3 | _] < [0, 3, 4 | _]

where "<" here means "less specific than." This partial ordering fits nicely together with the underlying Prolog engine. A pattern which matches, say, [0, 3, 4 | _] will also match [0, 3 | _], but not necessarily the other way round. If, e.g., the following generative clause,

p(*x) in *prog:– q(*x, *y) in *prog, . . .,

belongs to a program indexed [0, 3, 4 | _], we compile it as follows.

p([0, 3, 4 | _], Current, X):– q(Current, Current, X, Y), . . .

A query "p(*x) in [0, 3, 4]" is then compiled into "p([0, 3, 4], [0, 3, 4], X)" which can be safely executed by the Prolog system. Only clauses in programs whose indices are equal to or less specific that [0, 3, 4 | _] can be involved in the proof.

So clearly, this implementation is efficient concerned with space. With respect to execution speed, it should be noted that all additional meta-level control takes place in the head of compiled clauses and hence is treated in an optimal way by the Prolog system.

The following expressions can be used in right-hand-side program parts.

[*rule*, . . .] — a program of new rules

[*rule*, . . .] & *prog — extension in the style of asserta

*prog & [*rule*, . . .] — extension in the style of assertz

Furthermore, we can define programs excluding specified rules:

*prog \ [*rule-index*, . . .]

Rules to be excluded are referenced by rule indices created (optionally) when the rules are compiled. Writing "*i is *rule*" instead of just "*rule*" in the program part where it is defined will instantiate *i to a unique integer number. If for example a rule,

p(*x) in *prog:– . . .

is part of program [3, 4 | _] and assigned the rule index 117, it compiles into the following.

p([3, 4 | _], Current, X):– not locked(117, Current), . . .

If later, the rule is excluded from, say, program [3, 4, 7 | _], the following fact is added to the program data base.

locked(117, [3, 4, 7 | _])

Our solution to the semantic problems due to backtracking into retracted code is straightforward. Compiled clauses are never removed from the database during program execution.

The program indexing method is well-suited for programs calling themselves recursively on other programs in a not too deeply nested hierarchy, the maximum depth giving an upper limit for the size of the program indices. A worst case example is a simulation program which models time by gradually adding and removing clauses from a developing program. For each time step the length of the current program index is increased by one!

The implemented system is coupled with a meta-level extension of definite clause grammars called *generative clause grammars*. The impact of a grammatical meta-level on the descriptions of programming languages is described in (Christiansen, 1990). Based on additional techniques not described here, the implemented system can also handle generic predicates and nonterminals, which means that variables can be used anywhere predicates and nonterminals are expected. The source text for the system written in a standard Prolog notation is available on request from the author.

The structure sharing representation of hierarchies of programs has much in common with Bacha's (1987, 1988) more efficient compiler for MetaProlog although in the present work implemented in Prolog relying on the partial ordering of terms given by their degree of instantiation. Structure sharing as a general technique is treated in (Christiansen, 1989b).

7 Summary

We have shown that the intuitively simple, declarative semantics of definite clause programs can be generalized to meta-level programming. This is based on a syntactic combination of a goal and the program in which it is being proved, the meta-goal, which is the central concept in our language of generative clause programs, GCP. The well-known results with a fixpoint characterization of the least Herbrand model have been generalized to GCP, here the Herbrand models are sets of meta-goals. Thus we get a little step closer to an intuitive and declarative understanding of meta-*programming* — in addition to the operational understanding of meta-*interpretation*.

We were able to present a very simple interpreter and a compiler for the language, which, however only can execute meta-goals with completely given program parts. We have not yet provided a proper generalization of SLD-resolution (cf. Lloyd, 1987) for the full GCP language and a more deep study of the procedural properties of GCP is a subject for future research.

The background for the present work is the author's work on grammatical meta-levels in programming language syntax (Christiansen, 1985, 1988, 1990). There is a clear analogy between such grammatical descriptions and meta-level logic, the syntax trees of the former are isomorphic to the proof trees of the latter (Christiansen, 1989a).

References

Bacha, H., Meta-level programming: A compiled approach. *Logic Programming, Proceedings of the fourth international conference*, ed. Lassez, J.-L., MIT Press, pp. 394–410, 1987.

Bacha, H., MetaProlog design and implementation. *Logic Programming, Proceedings of the fifth international conference*, ed. Kowalski, R.A., Bowen, K.A., MIT Press, pp. 1371–1387, 1988.

Bowen, K.A., Meta-level programming and knowledge representation. *New Generation Computing* 3, pp. 359–383, 1985.

Bowen, K.A. and Kowalski, R.A., Amalgamating language and meta-language in logic programming. *Logic Programming*, Clark, K.L. and Tärnlund, S.Å., eds., pp. 153–172, Academic Press, 1982.

Bowen, K.A. and Weinberg, T., A meta-level extension of Prolog. *IEEE Symposium on Logic Programming*, pp. 669–675, Boston, 1985.

Bratko, I., *Prolog, Programming for artificial intelligence.* Addison-Wesley, 1986.

Christiansen, H., Syntax, semantics, and implementation strategies for programming languages with powerful abstraction mechanisms. *Proc. 18th Hawaii International Conference on System Sciences*, vol. 2, pp. 57–66, 1985.

Christiansen, H., The syntax and semantics of extensible languages. *Datalogiske skrifter*, no. 14, Roskilde University Centre, 1988.

Christiansen, H., Extensible logic for extensible languages, *Datalogiske skrifter*, no. 21, Roskilde University Centre, 1989a.

Christiansen, H., Structure sharing in incremental systems. *Structured Programming*, vol. 10/4, pp. 169–186, 1989b.

Christiansen, H., A meta-level extension of definite clause programs, *To appear.* 1990.

Colmerauer, A., *Les grammaires de metamorphose*, Groupe d'Intelligence Artificielle, Université de Marseilles-Luminy, 1975. Appears as "Metamorphosis grammars"in *Lecture Notes in Computer Science* 63, pp. 133–189, 1978.

van Emden, M.H. and Kowalski, R.A., The semantics of predicate logic as a programming language. *Journal of the ACM.* Vol. 23, pp. 733–742, 1976.

Hill, P.M. and Lloyd, J.W., Analysis of meta-programs. *Proc. of Workshop on Meta-Programming in Logic Programming* (ed. Lloyd, J.W.), pp. 27–42, Bristol, 1988.

Lloyd, J.W., *Foundations of logic programming*, Second, extended edition. Springer-Verlag, 1987.

Subrahmanian, V.S., Foundation of meta-logic programming. *Proc. of Workshop on Meta-Programming in Logic Programming* (ed. Lloyd, J.W.), pp. 53–66, Bristol, 1988.