Meta-programming in DLP

A. Eliëns

Vrije Universiteit, Department of Mathematics and Computer Science

De Boelelaan 1081, 1081 HV Amsterdam The Netherlands

email: eliens@cs.vu.nl

Abstract

We will describe the use of meta-programming in the distributed logic programming language DLP. Also we will discuss how to extend the language with a class/meta-class concept, to allow for reflective capabilities.

1 Introduction

Prolog offers powerful features for meta-programming. These features may be used for extending the language to incorporate other programming paradigms, such as for example the object oriented paradigm, or to embed special purpose inference procedures in the language. However, the penalty of using meta-programming to extend the language in this way is an obvious loss of efficiency.

We have developed the distributed logic programming language DLP, combining logic programming with object oriented features and parallelism. Our approach allows for an efficient implementation since the computation model of DLP directly supports the features derived from its underlying paradigms.

Nevertheless, the language DLP supports meta-programming in the style of Prolog. For example, meta-programming may be used to implement a restricted form of *and*-parallelism.

Concluding our paper, we will discuss the possible wyas to include reflective capibilites by extending our (prototype-based) object model to one based on a class/meta-class concept.

Our interest is motivated by our current efforts to re-implement DLP in Active C++, an extension of C++ that supports active classes and communication by rendez-vous.

2 DLP – a language for distributed logic programming

The language DLP is an extension of Prolog with constructions for parallel object oriented programming. Special expressions are introduced for dealing with non-logical variables, object creation, and engaging in a rendez-vous.

Object declarations An object declaration in DLP contains the names of the non-logical variables of the object and a definition of the two types of clauses, viz. constructor clauses and method clauses.

An object declaration in DLP has the form

object

```
object name {
var variables.
clauses
}
```

Both *object* and *var* are keywords. The variables declared by *var* are non-logical variables that may be assigned values by a special statement.

Objects act as prototypes in that new copies may be made by so-called *new* statements. Such copies are called *instances*. Each instance has its private copy of the non-logical variables of the declared object. In other words, non-logical variables act as instance variables.

A clause is considered a constructor clause when the predicate name of the head of the clause is identical to the name of the declared object. A method for an object is defined by all clauses having that method name as the predicate name of their head. The clauses are written in standard Prolog format, but may also contain as goals the special forms listed below.

Statements The special forms include:

- $v := t$ - to assign (the value of) the term t to the non-logical (instance) variable v
- $O = new(c(t))$ - to create a new instance of the object c and to start the constructor process evaluating the goal $c(t)$, O thereafter refers to the newly created object
- $O!m(t)$ - to engage in a rendez-vous with the object to which O refers, for evaluating the method call $m(t)$
- $accept(m_1, ..., m_n)$ - to state the willingness to engage in a rendez-vous by accepting one of the methods $m_1, ..., m_n$

When assigning a term t to a non-logical variable, all non-logical variables in the term are first replaced by their values. The same applies when unifying two terms.

Inheritance The language DLP supports (multiple) inheritance. The declaration for an object *name* inheriting from an object *base* looks as follows:

inheritance

```
object name : base {
var variables.
clauses
}
```

This declaration will result in adding the non-logical variables and clauses declared for the object *base* to those of the declared object. For an example of the use of inheritance to model distributed problem solving see [Eliëns, 1991].

DLP is a distributed language in the sense that it supports processes, communication between processes and is able to handle failure arising in a communication. The distinguishing feature of DLP, compared with other approaches at combining logic programming with object oriented features and parallelism, is its support for multi-threaded objects and the possible occurrence of distributed backtracking over the results of a rendez-vous. See [Eliëns, 1992].

The execution model: objects, processes and backtracking

An execution model for DLP must give an account of the behavior of objects and processes and must explain how process creation and communication interact with backtracking.

Objects serve as a means for modularization and provide protection of (local) data. Non-logical variables may be used to store the data encapsulated by an object. The clauses defined for an object act as methods, in that they have exclusive access to these data.

Processes are created on the occasion of creating a new object, and when requesting a rendez-vous. When a new object is created, the constructor process, defined by clauses for the predicate with the name of the object, is started. When a rendez-vous is requested by a method call, a process is started for evaluating the method call. Creating a new process for evaluating such a goal is necessitated by our approach to distributed backtracking. Both the constructor process and the processes started for evaluating a goal by an object are said to refer to that object.

Distributed backtracking may occur when multiple answer substitutions result from evaluating a method call. As an example, a method call to the object declared below may generate an indefinite number of solutions.

nat

```
obj nat {
number(0).
number(s(X)) :- number(X).
}
```

The constructor clause for the object states that any call to numwill be accepted.

Evaluating the goal

```
:- O=new(nat()), O!num(X), write(X), fail.
```

results in printing all natural numbers, eventually. The process created for evaluating num(X) backtracks each time that backtracking over O!num(X) is tried due to the occurrence of fail in the goal.

3 Meta-programming in Prolog

In Prolog, programs may be treated as data. To this end Prolog offers a number of so-called *meta-predicates* that allow to inspect and manipulate terms and clauses. As examples of such meta-predicate we mention the predicates *not*, the predicate *call* that usually is omitted as illustrated in the example below, and the predicate *clause* that may be used to enumerate all the clauses of a program.

Control structures Despite the declarative semantics of logic programming, Prolog programmers often rely on control structures that are reminiscent of control structures encountered in traditional imperative languages. As an example, we show the implementation of a conditional branching operator.

cond

```
A → B :- A, !, B.
```

The clause defining the implication arrow enforces that the condition A is only evaluated once.

A naive meta-interpreter A goal of the form $clause(A, B)$ may be used to select all clauses of which the head unifies with A and the body with B. When both A and B are variables the goal $clause(A, B)$ results in enumerating all clauses, binding A to the head and B to the body of the clause currently pointed at. For clauses that represent facts, B will be bound to true.

An example of using this facility is given by a simple (meta) interpreter for Prolog written in only a few lines of Prolog. C.f. [Sterling, 1986].

backward

```
backward(true).
backward((A,B)) :-
      backward(A),
      backward(B).
backward(A) :-
      clause(A,B),
      backward(B).
```

The case analysis performed by the interpreter is straightforward. The goal *true* succeeds immediately. A conjunctive goal is split into parts which are successively evaluated. An atomic goal is solved by looking for a clause of which the head matches the goal, and solving the body of that clause.

4 Meta-programming in DLP

Actually, for DLP we have implemented the synchronous rendez-vous operator in terms of the primitives of an asynchronous rendez-vous mechanism. The synchronous rendez-vous is implemented by the clause:

```
O!G :- Q = O!G, Q?.
```

As explained below, a synchronous rendez-vous consists of two parts, the creation of a process to evaluate a method call and the request for an answer.

An asynchronous rendez-vous mechanism For creation of the evaluating process, one may use the special statement

- $Q = O!G$

to request the object to which O refers to create a process to evaluate the goal G. The variable Q thereafter refers to that process. For requesting the result we have introduced the goal

- $Q?$

that asks the process, referred to by Q, for an (other) answer.

And-parallelism The decomposition of the rendez-vous allows to define and-parallelism in a rather straightforward way, as

```
A&B :- Q = self!B, A, Q?.
```

where *self* refers to the object evaluating the goal $A\&B$. Such goals may however occur only in passive objects, since only passive objects allow internal concurrency.

An advantage of this approach is that the programmer may restrict the cases where parallel evaluation occurs by imposing extra conditions (cf. [DeGroot, 1984]) as in

```
A&B :- ground(B),!, Q = self!B, A, Q?.
A&B :- A, B.
```

where splitting of a new process is allowed only when B is ground. Note that the cut in the first clause is used to avoid unwanted backtracking over the second solution of A&B.

Overriding Each object defined in DLP inherits its basic functionality from an object called *boot*. However, system-predicates may be redefined as to allow for specializations of these predicates. As an example, the synchronous rendez-vous operator may be redefined according to a users preferences. Note however that, since a redefinitions has effect only within the scope of the object, the modification does not affect the object that is asked to engage in a rendez-vous. In contrast, when overloading the dereferencing operator in C++ the behavior of the object receiving a call to a member-function is affected. Cf. [Stroustrup, 1991].

5 Discussion

Our approach to embedding reflective capabilities in DLP is not of a fully fledged object oriented nature, since until now only programs consisting of clauses may be created as first-class data but not objects consisting of non-logical variables and clauses.

In our prototype described in [Eliëns, 1992] we have experimented with the conversion between objects and terms. Converting a term to an object-definition is a trivial matter. However, it seems rather difficult to convert an active object, or an object busy with evaluating a method call, to a term. For a truly passive object, that is an object that is not active in any way, such a conversion is possible.

Taking these restrictions into account it seems, however, more elegant to use a class-based approach since what we are interested in is after all the description or declaration of an object, in other words its class.

¿From [Malenfant et al, 1989] we know that the class/meta-class notion derived from [Cointe, 1987] is also relevant to distributed logic programming. In the near future we will explore how to apply this approach to DLP.

References

[Cointe, 1987] P. Cointe, *Metaclasses are first class: the ObjVLisp Model,* in: Proc. OOPSLA'87, N. Meyrowitz (ed.), ACM Sigplan Notices 22 (12) (1987) pp. 156-167

[Eliëns, 1991] A. Eliëns, *Distributed Logic Programming for Artificial Intelligence,* AI Communications Vol. 4 No. 1, 1991, pp. 11-21

[Eliëns, 1992] A. Eliëns, *DLP - A language for Distributed Logic Programming,* Wiley (1992)

[Eliëns and Visser, 1992] A. Eliëns and C. Visser, *Active C++ — active classes and communication by rendez-vous,* Technical Report Vrije Universiteit (to appear)

[Malenfant et al, 1989] J. Malenfant, G. Lapalme and J. Vaucher, *ObjVProlog: Metaclasses in Logic,* in: Proc. ECOOP'89, Cambridge University Press (1989) pp. 257-269

[Sterling, 1986] L. Sterling, *Expert System = Knowledge + Meta-Interpreter,* Report, The Weizman Institute of Science, Israel

[Stroustrup, 1991] B. Stroustrup, *The C++ Programming Language,* Addison Wesley (1991) 2nd edn.