3

Distributed Logic Programming

Distributed Logic Programming combines logic programming, object oriented programming and parallelism, which may be characterized by the pseudo-equation

$$DLP = LP + OO + ||$$

The language DLP can be regarded as an extension of Prolog with object declarations and statements for the creation of objects, communication between objects and the destructive assignment of values to non-logical instance variables of objects.

3.1 OBJECT DECLARATIONS

Object declarations in DLP have the following form:

```
:- object name.
var variables.
clauses.
:- end_object name.
```

where *object* and *end_object* are directives to delimit an object. Variables declared by *var* are non-logical variables; they 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.

Dynamically, a distinction is made between active objects and passive objects. Active objects must explicitly be created by a *new* statement. Syntactically, the distinction between active and passive objects is reflected in the occurrence of so-called constructor clauses in the declaration of active objects. Constructor clauses are clauses of which the head has a predicate name identical to the name of the object in which they occur. Constructor clauses specify an object's own activity. Other clauses in an object declaration may be regarded as method clauses, specifying how a request to the object is handled. Passive objects only have method clauses.

3.2 STATEMENTS

DLP extends Prolog with a number of statements for dealing with non-logical variables, the creation of objects and the communication between objects. These statements may occur as a goal in the body of a method.

Non-logical variables. For assigning a term t to a non-logical variable x the statement

```
x:=t
```

is provided. Before the assignment takes place, the term t is simplified and non-logical variables occurring in t are replaced by their current values. In fact, such simplifications take place for each goal. DLP also supports arithmetical simplification.

New expressions. For dynamically creating instances of objects the statement

$O := new(c)$

is provided, where c is the name of a declared object. When evaluated as a goal, a reference to the newly created object will be bound to the logical variable O. For creating active objects the statement

$O := new(c(t_1, ..., t_n))$

must be used. The activity of the newly created object consists of evaluating the constructor goal $c(t_1, ..., t_n)$, where c is the object name and $t_1, .../t_n$ denote the actual parameters. The constructor goal will be evaluated by the constructor clauses.

Method calls A method call is the evaluation of a goal by an object. To call the method m of an object O with parameters $t1, ..., t_n$ the statement

$O < -m(t_1, ..., t_n)$

must be used. It is assumed that O is a logical variable bound to the object to which the request is addressed. When such a goal is encountered, object O is asked to evaluate the goal $m(t_1, ..., t_n)$. If the object is willing to accept the request then the result of evaluating $m(t_1, ..., t_n)$ will be sent back to the caller. After sending the first result, subsequent results will be delivered whenever the caller tries to backtrack over the method call. If no alternative solutions can be produced the call fails. Active objects must explicitly interrupt their own activity and state their willingness to accept a method call by a statement of the form

$accept(m1, ..., m_n)$

which indicates that a request for one of the methods $m_1, ..., m_n$ will be accepted.

Inheritance An essential feature of the object oriented approach is the use of inheritance to define the relations between objects. Inheritance may be conveniently used to factor out the code common to a number of objects.

The declaration of an object *name* inheriting from an object *base* is:

```
:- object name : [base].
var variables.
clauses.
:- end_object name.
```

Multiple base objects are separated by "," in the declaration like:

$:-object\ \ name : [base_1, base_2, ..., base_n]$.

3.3 EXAMPLES

3.3.1 Hello World

The following is a simple DLP program, which only prints the text 'hello world'.

```
:- object hello_world1.

main:-
    format('Hello, World ~n').

:- end_object hello_world1.
```

Similar to Java, the predicate 'main' is the starting point of an object. If the program above is saved in a file 'helloworld1.pl', we can use the command 'dlpc

helloworld1.pl' to compile the DLP program. Note that the file extension of a PROLOG or DLP program is 'pl'. After compiling, we can use the command 'dlp hello_world1' to run the program. Note that we use the file name to compile a DLP program file, however, we use the object name to run a DLP program, for a program may consist of multiple objects.

The second 'hello world' example prints 20 times the text 'Hello World':

```
:- object hello_world2.

var count=0.

main:-
    repeat,
       format('Hello World ~w ~n',[count]),
       ++count,
    count >= 20,
    !.

:- end_object hello_world2.
```

The program above uses a non-logical variable 'count' as a counter in a repeat loop. If the goal '$count \geq 20$' fails, the program will backtrack to the start of the 'repeat' loop until the condition '$count \geq 20$' succeeds. The program above looks more like a procedural Java program instead of a declarative program. However, we can design a recursive program which behaves the same but without using the non-logical variable and the 'repeat' loop. We leave it as an exercise.

In the following, we show an object which prints the text 'hello', and another object which prints the text 'world'.

```
:- object hello.

var count=0.

hello:-
    repeat,
      format('hello ~w~n',[count]),
      sleep(500),
     ++count,
    count >= 10,
    !.

:- end_object hello.

:- object world.
```

```
var count=0.

world:-
    repeat,
      format('world ~w~n',[count]),
      sleep(250),
      ++count,
    count >= 10,
    !.

:- end_object world.

:- object hello_world3.

main:-
      _H := new(hello),
      _W := new(world).

:- end_object hello_world3.
```

The object 'hello_world3' uses the 'new' statement to create two threads. In order to see that these two threads run independently, we specify different 'sleep' times for each thread. The goal 'sleep(500)' is used to delay the thread for 500 milliseconds.

Observing the fact that the above-mentioned objects 'hello' and 'world' have the same program structure, we can implement a single object to achieve the same result. Moreover, in the following, we want to design a 'hello world' program in which multiple threads can share an independently running 'clock' thread. The 'hello' and 'world' threads print their text until the clock counts 10 :

```
:- object clock.

        var time = 0.

        get_clock(T):-
          T := time.

        set_clock(T):-
          time := T.

:- end_object clock.

:- object pulse.
```

```
pulse :-
    repeat,
      sleep(1000),
      clock <- get_clock(T),
      T1 is T + 1,
      clock <- set_clock(T1),
    T1 > 10,
    !.

:- end_object pulse.

:- object hello4.

hello4(Text):-
    repeat,
      clock <-get_clock(Time),
      format('~w ~w~ n', [Text,Time]),
      sleep(500),
    Time >= 10,
    !.

:- end_object hello4.

:- object hello_world4.

main:-
      _C := new(pulse),
      _H := new(hello4(hello)),
      _W := new(hello4(world)).

:- end_object hello_world4.
```

The object 'clock' provides the two methods *get_clock* and *set_clock* for getting and setting the clock pulse counter. The object 'pulse' simulates a clock by incrementing a counter after each repeat step.

3.3.2 File I/O

This is an example of file I/O operations in DLP.

```
:- object pxfile.

var x, y, z.
```