On the use of semantics:

Extending Prolog to a Parallel Object Oriented Language

A. Eliëns
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

This contribution intends to demonstrate how the study of semantics influenced both the implementation and design of a parallel logic programming language, which extends Prolog to a parallel object oriented language.

Classes play a central role in the language. An instance of a class may be asked to evaluate a goal. On creating a process to evaluate a goal the invoking process receives a pointer to that process. The answer substitutions resulting from the evaluation of the goal can, in an asynchronous fashion, be collected by a resumption goal using this pointer. A synchronous communication construct is provided. Communication is blocking, the input side however is allowed to backtrack until an input term is found that unifies with the output term. Non-logical variables, local to an instance of a class, that can hold arbitrary terms as values, may be used to record the state of an instance. Synchronisation between processes referring to an instance of a class can be achieved with an answer statement that allows to postpone the evaluation of a goal until the state of that instance satisfies a particular condition. Instances of classes, and processes evaluating goals, can be allocated explicitly on a processor node, to effect a proper distribution of the computation. Two forms of inheritance are provided, one consists of using clauses of another class, the other consists of copying the non-logical variables of another class into each instance of the class.

- The question is the story itself, and whether or not it means something is not for the story to tell -
Paul Auster, The New York Trilogy.

1. Introduction

Semantics as other fields of science, knows a gap between theory and practice. I've experienced some of the difficulties in bridging this gap when I made an attempt to apply the techniques of formal semantics to the description of expert systems. Yet, as a tribute to Prof. J.W. de Bakker, who was my guide in this field of science, I would like to sketch how the study of semantics influenced both the implementation and design of a (prototype) parallel Prolog system. This work concerns an effort to augment (standard) Prolog with constructs that allow parallel object oriented processing.

The structure of this paper is as follows: The major part of the paper will be devoted to a description of the language constructs for process-creation, communication and synchronization. Then I will informally describe a continuation semantics derived from [Al86] that underlies the implementation of the sequential Prolog interpreter. The design of a unifying communication construct was influenced

The work in this document was conducted as part of the PRISMA project, a joint effort with Philips Research Eindhoven, partially supported by Esprit 415.

by my studies of the semantics of the synchronous communication construct of Occam [In84]. I will motivate the choice for a particular variant by discussing some of the alternatives. Lastly, I will comment on the current status of the system and return the question back to the field that has inspired me: to provide a formal description of the language extensions proposed.

2. EXTENDING PROLOG TO A PARALLEL OBJECT ORIENTED LANGUAGE

Languages such as Concurrent Prolog and Guarded Horn Clauses have made the choice for implicit parallelism, which means that the parallelism is exploited by the implementation and as such hidden to the programmer. Cf. [Bo89], [Co87]. In contrast, the language extensions that will be proposed provide the programmer with explicit control over the creation (and allocation) of new processes, and with the means to effect synchronisation and communication between processes.

The idea behind these extensions is to lift the constructs for parallelism as present in a parallel object oriented language like POOL [Am87] to a Prolog-like language. A class definition in POOL contains a description of the behavior of an object, a collection of data and methods that operate upon these data. For reasons of protection only methods are allowed access to these data. Each object, that is instance of a class, is associated with a process, in that it may either answer method calls or perform actions on its own. An object may postpone answering certain method calls, dependent upon its state, by means of an answer statement that sets the answer list, the list of methods that are answerable.

Classes, also, play a central role in the language to be proposed. The evaluation of a goal, by an instance of a class, is to some extent analogous to answering a method. A difference with POOL, however, is that in our case the identification of an instance of a class and a process does not hold. A process is created, on explicit demand, for evaluating a goal. The class instance merely provides the functionality needed for evaluating the goal, and possibly some protection and synchronisation.

Examples will be given that show how logic programming and (parallel) object oriented programming may be combined. It turns out, however, that some of the behavior associated with object oriented programming can be achieved without the synchronisation mechanism provided by classes. An Occam-like communication mechanism, extended to deal with unification, seems to suffice for these cases. Cf. [In84], [PN84].

2.1. Classes

Simple classes are just modules, that is collections of clauses, that define the functionality of a process evaluating a goal. A process can be created from (an instance of) a class by asking it to evaluate a goal. Such a process is said to refer to an instance of a class. Each instance may have a multiple of processes referring to it. The evaluation of a goal in itself is sequential, unless other processes are explicitly created.

The true virtue of a class comes into view when non-logical variables are used. Each instance of a class may contain local non-logical variables, that can hold arbitrary terms as values. Mutual exclusion between the evaluation of goals by processes referring to an instance is needed to avoid simultaneous access to these variables. For the purpose of synchronisation, moreover, an answer-statement is provided, to allow an instance to postpone the evaluation of certain goals dependent on its state, as expressed in the values of its non-logical variables. A class thus extends a simple class by the possibility of local non-logical variables, mutual exclusion in the evaluation of goals, and the possibility to postpone the evaluation of goals dependent upon its state. Hence, unless only one instance of a class is used, these differences necessitate making a copy of a class containing non-logical variables before using it, whereas simple classes need not be copied since no values are shared over different invocations. Only when an answer-statement is encountered, mutual exclusion between goals takes effect. Class constructors, goals having the same name as the class, can be used to initialize an instance with respect to its local variables and its willingness to answer a query.

An example of the declaration of a simple class is given by

```
class list {

member(X,[X|_]).
member(X,[_|T]):- member(X,T).

append([],L,L).
append([H|T],L,[H|R]):- append(T,L,R).

}
```

This class contains no variables or answer-statement, and hence is just a collection of clauses. A class may use other classes by putting them in its use list. Only simple classes may be put in a use list, however, since the variables of the used class cannot be accessed directly.

The extensions to Prolog that allow parallel object oriented programming are first of all a construct for process creation and a means to collect the answer substitutions resulting from the evaluation of a goal. Secondly, a mechanism for synchronous communication will be treated. Next, it will be described how local, non-logical variables and answer statement can be used to govern the behavior of an object, that is its willingness to evaluate goals. Lastly, after dealing with issues of allocation and distribution, a simple form of inheritance will be discussed.

Since the primary intent here is to give an intuition for the mechanisms needed for parallel object oriented logic programming, and to motivate the constructs proposed by examples, the description of the constructs itself will be rather informal.

2.2. Process creation and resumptions

Processes created to evaluate a goal derive their functionality from a class in that the evaluation of the goal takes place by the clauses contained in that class. A goal statement of the form

`Q = new(`*classname*`)!G`

where `G` is an arbitrary term and `Q` a variable, starts up a process derived from the class *classname* to evaluate the goal `G`. The call is *asynchronous* because `Q` is bound to a pointer to the process evaluating the goal. This pointer, the value of `Q`, will for the lack of a better name, be called the *query-status*.

The query-status allows to process the answer substitutions resulting from `G` in the context from which the process for evaluating `G` was split off, by stating the *resumption* goal

`Q?`

which results in the evaluation of a (special) goal that effects the bindings in the current context. Conceptually, this goal amounts to backtracking over the possible solutions until no other solution is available. When backtracking occurs, all answer substitutions provided by `Q?` will be tried before backtracking over any goal preceding `Q?`.

Parallelism is achieved by the fact that the newly created process runs independently of the process that created it, as reflected in the statement sequence

```
Q = new(classname)!G,...,Q?
```

In between the creation of the process and stating the *resumption* goal to collect the answer substitutions, the invoking process can perform whatever action suitable. Backtracking over this sequence will not undo the creation of the process for `G`. When a goal preceding the sequence however gives rise to a new solution, a new process for evaluating `G`, possibly differently instantiated, may be created. The occurrence of a cut has effect only locally in the process in which it occurs, in restricting the answer substitutions.

This mechanism of process creation and resumption allows to define and-parallelism in a rather straightforward way, as

```
A&B :- Q = new(this)!A, B, Q?.
```

where `this` refers to the class evaluating the goal `A & B`. Note that, somewhat counter-intuitively, backtracking will be done first over `A` and then over `B`.

The advantage of this approach is that the programmer may restrict the cases where parallel evaluation occurs by imposing extra conditions (cf. [DG84], [HN86]) as in

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

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

A typical example of the usage of this kind of parallelism is the following, familiar, quicksort program.

EXAMPLE 1: *quicksort*

```
class sort {
use list.

sort([],[]).
sort([X|T],S):-
  split(X,T,L1,L2),
  ( sort(L1,S1) & sort(L2,S2) ),
  append(S1,[X|S2],S).

split(X,[],[],[]).
split(X,[Y|T],[Y|L1],L2):-
       X > Y,!,
       split(X,T,L1,L2).
split(X,[Y|T],L1,[Y|L2]):-
       split(X,T,L1,L2).

}
```

Each nonempty list is divided into two sublists, one with values less than the values in the other, that are sorted in parallel and then appended.

2.3. Communication and synchronisation

Simple classes, without non-logical variables, allow in combination with a construct for synchronous communication features associated with object oriented programming. Cf. [ST83], [PN84].

Communication between processes can take place by means of input and output statements over channel-like objects, similar as for example in Occam [In84], or Delta Prolog [PN84]. The difference with Occam, however, is that instead of assignment of a value on the input side a unification of the input and output term occurs on both sides. Moreover, whereas Occam requires the number of channels to be fixed in advance, an indefinite number of channels is allowed.

A new channel can be created by the statement

```
C = new(channel)
```

The evaluation of this statement results in binding the variable `C` to a special channel-like object.
An output statement is of the form

```
C!T
```

with `C` referring to a channel, and `T` to an arbitrary term.
An input statement has the form

```
C?T
```

with similar meanings of `C` and `T`. A simple example of a unifying communication is given by

```
C!f(a,X) & C?f(Y,b)
```

which results in the binding of `X` to `b` and of `Y` to `a`.

The communication is *synchronous*, in that the output side waits until there is an input goal such that the input and output term are unifiable, and similarly the input side waits until an output goal occurs. Communication however is *asymmetric* in that the output side waits until a unifiable input term is available, while the input side is allowed to backtrack once an output term is available. Such behavior can be observed with a goal like

```
C!f(X) & ( C?g(a) ; C?f(a) )
```

with ";" the standard Prolog disjunctive operator.

The motivation for allowing backtracking on the input side is illustrated by the following example of a counter.

EXAMPLE 2: *counter*

```
class ctr {

ctr(C):- run(C,0).

run(C,N):-
        C?inc(N),
        N1 = N + 1,
        run(C,N1).
run(C,N):-
        C?value(N),
        run(C,N).

run(C,N):-
        C?stop.

}
```

A typical example of the use of such a counter is

```
:-
     C = new(channel),
     Q = new(ctr(C)),
     C!inc(N),
     C!stop.
```

The call `Q = new(ctr(C))` initializes the instance of `ctr` to channel `C`, and is a shorthand for `Q = new(ctr)!ctr(C)`. Since both sides of the communication are blocking, either a catchall of the form `C?_` , where _ is an anonymous variable, must be present on the input side, or an explicit fail command `fail(C)`, that results in failure and backtracking on the output side, while it succeeds on the input side. Once a communication is successful no more backtracking will occur on the input side. When the output side backtracks only a new attempt at communication may be made.

The example above is adapted from [PN84] and was originally given in [ST83] to demonstrate how object oriented programming can be implemented by continuously running processes receiving a stream of messages.

There is an obvious resemblance with the communication construct proposed for Delta Prolog [PN84]. The possibility to create an arbitrary number of channels however allows for setting up a system of processes in a way that is hard to envisage in Delta Prolog.

As an example of a situation where the number of processes and channels between processes can grow indefinitely large I've taken the solution to generating primes as presented in the user-manual for POOL-T [Au85]. The idea, based on the sieve algorithm of Eratosthenes, is to implement each sieve as an independent process connected to a predecessor by means of an input channel and to a successor process by an output channel. The first process generates odd numbers, and sends it to the first sieve. On creation, a sieve waits for the first incoming number, which will be its prime number. For each subsequent number a sieve checks whether it is divisible by its prime. If this is not the case then the number is sent to the next sieve.

EXAMPLE 3: *prime generator*

```
class driver {

driver(I):-
        C = new(channel),
        Q = new(sieve(C)),
        drive(C,I).

drive(C,I):-
        C!I, J = I+2, drive(C,J).
}

class sieve {

sieve(C):-
        C?I,
        collect(I),
        C1 = new(channel),
        Q = new(sieve(C1)),
        run(I,C,C1).

run(P,Cin,Cout):-
        Cin?I,
        ( I//P != 0 -> Cout!I | true ),
        run(P,Cin,Cout).

collect(I):- write(I).

}
```

The program is started by the goal

```
:- Q = new(driver(3)).
```

The output can be collected by sending each prime with which a new sieve is initialized to a special process. The goal `I // P != 0` is evaluated by simplifying `I // P` to `I` *modulo* `P` followed by a test whether the result is unequal to zero.

2.4. Non-logical variables

The next language extension I wish to discuss concerns the use of non-logical variables local to an instance of a class to store persistent values.

Again, the counter may serve as an example.

EXAMPLE 4: *counter*

```
class ctr {
var num=0.

inc(N):- N = val(num), val(num) = val(num) + 1.
value(N):- N = val(num).

}
```

This example differs from the previous solution in that the object oriented behavior is not achieved by keeping the value of the counter as an argument in a tail-recursive loop, but as an explicit (non-logical) variable that can be updated by assignment. The declaration

```
var num=0.
```

creates such a variable, and initializes it to zero. The variable can be accessed by the expression `val(num)`. The convention is used that when `val(num)` occurs on the left hand side of an equality that the value on the right hand side is stored as the value of the variable `num`.

Communication with a counter of this kind, incrementing and asking for its value, does not take place by means of input and output statements over channels, but by using the parameters of a goal, as illustrated in

```
:-
   C = new(ctr),
   C!inc(N),
   C!value(N1).
```

Since `C` is bound to an instance of the class `ctr`, and not to a channel, `C!G` defined by

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

must be considered a synchronised version of remote goal evaluation. The goal `C!inc(N)` thus amounts to binding the variable `N` to zero, the value of the local variable `num`, with as a side-effect an increment of this variable by one. This example directly illustrates the analogy between answering a method call and evaluating a goal as hinted at in discussing POOL. Side-effects will not be undone on backtracking.

2.5. Mutual exclusion and answer statements

Typically, a counter is used by only one process. When an instance of a class is referred to by more than one process, mutual exclusion between the evaluation of goals must be provided, in that no two processes are simultaneously granted access to a non-logical variable of that instance. The example of a semaphore, given below, moreover shows also the need of *answer*-statements to postpone the evaluation of a goal.

EXAMPLE 6: *semaphore*

```
class sema {
var num=0.

sema():- run.
sema(N):- val(num) = N, run.

p():- val(num) = val(num) - 1.
v():- val(num) = val(num) + 1.

run:-
        ( val(num) == 0 -> answer(v) ; answer(p,v) ),
        run.
}
```

The constructor for `sema` causes the semaphore to loop over a conditional that tests the value of the non-logical variable `num`. When its value is zero only calls to `v()` will be answered, otherwise both `p()` and `v()` may occur.

An example of its use is given by

```
cs(S,G):- S!p(), G, S!p().

:- S = new(sema(1)), ( cs(S,g1) & cs(S,g2) ).
```

which enforces that the goals `g1` and `g2` will be executed in a non-overlapping fashion. Since the new semaphore is initialized with one, the first process that asks for `p()` must first ask for `v()` before the other process is granted the evaluation of `p()`.

Operationally, when an answer statement is reached, the evaluation of the current goal is suspended until a goal allowed by the answer list has been evaluated. When the evaluation of a goal not occurring in the answer list is asked for, the process asking for the evaluation is suspended. The process continues when an answer statement occurs that allows it, and moreover, it is first in the queue of (allowed) processes waiting for the evaluation of a goal. The interplay between processes evaluating a goal and the instance of the class to which they commonly refer is rather subtle. It is the instance of the class that decides which processes are to be suspended and which processes may be active in evaluating a goal. Only one process may be active in evaluating a goal. It stops being active either upon termination, or by evaluating an answer statement. In both cases that particular instance may grant the evaluation of goals, as long as they are allowed by the answer list.

2.6. Dining philosophers

Our pièce de résistance is an implementation of the solution to the problem of the *dining philosophers*, as given in [Au85].

Five philosophers must spent their time *thinking* and *eating*. When a philosopher wants to eat he must get hold of two forks. Since only five forks are available, each philosopher must await his turn. To avoid the situation where each philosopher sits waiting with one fork, only four philosophers at a time are allowed in the dining room.

Since a philosopher needs to know no more than his name, the dining room and his proper forks, after creation he may proceed to enter the cycle of thinking, entering the dining room, picking up his forks, eating and leaving the dining room to start thinking, again.

```
class philosopher {
var name.

philosopher(Name,R,Lf,Rf) :-
        val(name) = Name,
        proceed(R,Lf,Rf).

think :- N = val(name), write(thinking(N)).
eat:- N = val(name), write(eating(N)).

proceed(R,Lf,Rf):-
        think,
        R!enter(),
        Lf ! pickup(), Rf!pickup(),
        eat,
        Lf ! putdown(), Rf!putdown(),
        R!exit(),
        proceed(R,Lf,Rf).
}
```

A philosopher is admitted to the dining room when less than four guests are present, otherwise he must wait for one of his collegues to exit.

```
class room {
var occupancy=0.

room():-
        (
        val(occupancy) == 0 -> answer(enter) ;
        val(occupancy) == 4 -> answer(exit) ;
        answer(enter, exit)
        ),
        room().

enter():- val(occupancy) = val(occupancy) - 1.
exit():- val(occupancy) = val(occupancy) + 1.

}
```

Forks can only be picked up and then put down.

```
class fork {

pickup().
putdown().

fork() :-
        answer( pickup ),
        answer( putdown ),
        fork().

}
```

The ceremony is started by assigning the philosophers their proper forks and showing them the dining room. The details of their initiation will, for reasons of space, be omitted.

The example fully demonstrates the synchronisation enforced by answer statements. Such behavior could not be effected by using simple classes and the synchronous communication construct presented earlier. Much worse, the synchronisation capability of classes makes such a communication construct superfluous. My personal preference, however, is to use the communication construct in cases where the sophisticated synchronisation provided by the answer statement is not demanded.

2.7. Distribution and allocation

In the examples given no attention has been paid to the issue of allocating class instances and the actual distribution of computation over the available resources.

When a new instance of a class is created, it can be allocated on a particular processor node, by a statement of the form

`O = new(`*classname*`)@N`

where `N` is the number of a processor node. Also it is possible to split of a process to evaluate a goal on a particular node by the statement `G@N` for `G` a goal, and `N` a node number. Conceptually, the meaning of a goal `G@N` is given by the clause

```
G@N :- O = new(this)@N, Q = O!G, Q?.
```

Since allocation is dependent on assumptions concerning the parallel machine on which the system is implemented these assumptions will be made explicit first.

The parallel processor for which the system is intended, is assumed to have a limited number (less than 100) of processor nodes that are connected with each other by a packet switching network, such that the distance between each node never exceeds a fixed number (3 or 4) of intermediate nodes. For reasons of optimal utilization such machines are considered to support coarse grain parallelism, which means that the ratio of communication and computation must be in favor of the latter.

The programmer knows each processor just by its number, $0 \cdots n-1$, for n processors. However to permit a more refined strategy of allocating processes and resources the language also allows *node expressions* from which a node number can be calculated. Apart from viewing the network as a sequence the programmer can index the nodes as a tree, as in

$$I_0 : I_1 : \cdots : I_n$$

which denotes, with the branching degree (by default) fixed to two, the processor associated to the node in the tree reached by following the path $I_1,...,I_n$ from I_0. The association of processor numbers to nodes of the tree is done by counting the nodes of tree breadth first. For example the expression $0:1:2:1$, giving the path 1,2,1 from 0, results in processor number 9.

As an example of how node expressions can be used to distribute the load of computation over the available processors consider the following variant of the quicksort program presented earlier.

```
class sort {
use list.

sort(L,R):-  sort(0,L,R).

sort(N,[],[]).
sort(N,[X|T],S):-
  split(X,T,L1,L2),
  ( sort(N:1,L1,S1)@N:1 & sort(N:2,L2,S2)@N:2 ),
  append(S1,[X|S2],S).

  . . .

}
```

When splitting off two processes for sorting the sublists, the processes are allocated on the successor nodes of the current node, as long as sufficient processors are available. More refined strategies may be encoded by including tests on the length of the lists.

The processor topology may also be viewed as a matrix with a certain width (say 4 for 16 processors). The programmer can index this matrix by expressions of the form

$$N_1 \# N_2$$

which allows to distribute the load over the available processors by moving, for instance north from $N_1 \# N_2$ to $N_1 \# N_2 + 1$, over the matrix. Such indexing might be useful for the example of grid computation given in [BL86].

2.8. Inheritance

Two forms of inheritance are provided. A class may put another class in its *use* list. The clauses of that class are then used whenever the class itself provides no clauses for a goal. When such clauses do occur the clauses of the other class will not be used for evaluating that goal. When non-logical variables are declared however, it is usually not possible to use clauses from that class, since a reference to these variables has no meaning for the class in which they are used. A simple solution to this problem is to allow a class to copy the non-logical variables of the other class to each instance of itself. Such course of action is taken for all classes in the *isa* list of a class.
For instance, the declaration

```
class b { var num. }

class a { isa b.  }
```

effects that all instances of `a` have a non-logical variable `num`.

The notation

```
class a : b { ... }
```

is used as a shorthand for

```
class a {
isa b.
use b.
 ...
}
```

and allows to define a semaphore by declaring

```
class number { var num=0. }

class counter : number {

get(N):- N = val(num).
put(N):- val(num) = N.
inc():- val(num) = val(num) + 1.
dec():- val(num) = val(num) - 1.

}

class semaphore : counter {

semaphore(N) :- put(N), run.

p():- dec().
v():- inc().

run:-
        get(N),
        ( N == 0 -> answer(v); answer(p,v) ),
        run.
}
```

Hence a `semaphore` is a kind of `counter`, which itself is a kind of `number`. Whenever multiple variables with the same name and different initializations occur, the result is undefined.

3. A SEQUENTIAL INTERPRETER FOR PROLOG

The implementation of the interpreter is based on a continuation semantics for Prolog given by [Al86]. The requirement imposed on the implementation language by this semantics is the possibility to treat functions as first class objects. For this reason, and several others, POOL-X was the natural choice for implementing the prototype.

The main idea of the semantics given in [Al86] is that a theory, that is a list of clauses, is compiled prior to a query into a *database* function that delivers a list of answer substitutions. Once a proper definition of composition of these database functions was found the treatment of classes using other classes did not present any serious problems. Although the intent of this section is to demonstrate how to derive an implementation from a formal semantic definition I will neither present a formal (continuation) semantics, nor the (POOL-X) code derived from it; instead I will merely give an outline of the approach followed in implementing the interpreter.

The implementation is term-oriented, in that all objects that may occur in the language, such as instances of classes, clauses etc, are treated as terms. The type T, for terms allows all the types connected with these objects as its underlying types. A distinction is made between a simple term and a compound term which is a term with a comma as a function-symbol. Compound terms are used to make lists of terms. This characterization of a term allows also clauses and theories to be treated as (lists of) terms. A Prolog clause, for instance, is a term with function symbol ":-" and as arguments a single term for the head and a possibly compound or empty term for the body.

Some preliminaries for dealing with the renaming of variables and substitutions must be given. When a new clause is tried to solve a goal, to rename the variables of the clause, an environment function of type $V \to V$ is needed, where V is the type for variables. Given such a function e the function call $mape(e,t)$ gives as a result the term t with all variables renamed according to e. For substitutions s of type $S = V \to T$ a similar function $maps$ is used to effect the substitution s on a term.

The trick in setting up a continuation semantics is the following: to define a function that performs a certain task, think of what can be done in one step, formulate what must be done afterwards, and do the step with as a continuation that what must be done to complete the task.

The continuations used for the Prolog interpreter have type $C = N \times N \times S \to T$. The first (integer) argument is used to keep track of the depth of a derivation, the second argument is used as a count of the number of variables used, and the third argument, a substitution, is needed to effect the bindings that resulted from evaluating a goal thus far.

Before the evaluation of a goal can be handled the clauses in a program must be compiled to a database-function of type

$$D = Q \times T \times C \times N \times N \times S$$

A function call to a database function d has the form $d(q,t,c,m,n,s)$, where q is the query-status that handles the dynamic aspects of the evaluation of a goal such as checking for cuts, communication with other processes and the like, t the actual goal to be evaluated, c the continuation containing what must be done after the goal t is solved, m the depth of the evaluation, n the number of variables in use and s the substitution built up thus far.

For compiling a theory into a database function , an auxiliary continuation type $K = D \to D$ is needed. To define the function *compile* of type $T \times D \times K \rightarrow D$, called by *compile*$(th,d,k)$, three cases must be distinguished. When the theory to be compiled is empty the result is just d, when the theory *th* is compound then a continuation, to compile the rest of the theory, must be prepared and the first clause must be compiled with the continuation just prepared. Compiling a single clause results in the creation of a new database function

$$newd := fn\ (q:Q,\ t:T,\ c:C,\ m,n:N,\ s:S):T$$

and involves setting up a new environment to rename the variables in the clause according to n, a new substitution that sets the variables in the clause to undefined, and the creation of a continuation for evaluating the body of the clause. Then the result of the function *newd* is the result of appending the result of unifying the goal t with the head of the clause with as a continuation the evaluation of the body of the clause, to the result of applying the given database function d to the arguments of *newd*. This way all alternative solutions provided by the theory are searched for. The function *compile* then results in applying the continuation k to *newd*, to allow for the compilation of any succeeding clauses.

The initial call to *compile* is, given a list of clauses *th*, *compile*(*th*,*nilD*,*nilK*). Since *nilD* will be the first database function applied for any goal, the result of *nilD* must naturally be *nil*. The continuation *nilK* also must be empty, but must act as an identity on D.

Classes are allowed to import the theory of other classes via their *use* list. A naive approach would consist of combining the theories of each class into one and then to compile the combined theory. However, composition of database functions is easily defined, and allows a seperate compilation of classes. The function $fn\ co(d_1,d_2:D):D$ results in a database function

$$d := fn(q:Q,\ t:T,\ c:C,\ m,n:N,\ s:S):T$$

defined as the result of applying d_1 to the arguments of d when this is not *nil* and the result of applying d_2 to the arguments of d otherwise.

The function *newd* created when compiling a clause contained a call to a function *unify* with as continuation the evaluation of the body of the clause. The function

$$fn\ unify(a,b:T,\ c:C,\ m,n:N,\ s:S):T$$

must, when one of the terms a or b is a variable, create a substitution *news* to update the given substitution. When the variable is still unbound then the contination c is called with the substitution *news*, otherwise the value bound to the variable must be unified with the other term. When only non-variable terms are involved either a test on equality is performed, in case both terms are constants or, in the case of function terms, a unification of the arguments is tried. Both cases, when successful, result in executing the continuation.

The continuation created for the body of the clause contains a recursive call to a *query* function, from which the database function will initially be called. The function *query* is of type similar to a database function, its main purpose however is to deal with compound goals and to keep account of cuts. The function

$$fn\ query(q:Q,\ t:T,\ c:C,\ m,n:N,\ s:S):T$$

results in the continuation $c(m,n,s)$ when the goal is empty. For a compound goal a continuation for evaluating the rest of the goal is created and the database function, stored in the query-status q, is applied to the first term of the goal. For a simple goal the creation of a continuation is omitted.

When a cut occurs as a goal, the query-status is notified of the occurrence of a cut at that particular depth. The database function produced by *compile* includes a test, consulting the query-status q, to prevent backtracking when a cut has occurred.

For a given goal *t* the function *query* is called as in *query*(*q*,*t*,*yes*(*q*,*t*),0,*n*,*starts*), where *n* is the number of variables occurring in *t* and *starts* the substitution that sets each variable to undefined. The initial continuation is given by applying the function *yes* to *q* and *t*. This function first stores the list of variables occurring in *t* into the query-status *q* and then creates a continuation that reports, if the evaluation of the goal is successful, the substitution values of these variables to *q*. This allows to ask the query-status for the answer substitutions resulting from the evaluation of the goal.

For the occasion I've taken the liberty to give a semantics in prose, giving only the gist of the semantics on which this implementation is based. Nevertheless, I hope to have demonstrated the usefulness of semantics from an implementation point of view, to which I may add the encouraging fact that the actual implementation of what is described is of about the same number of lines as the verbiage representing it.

4. Design issues

The prototype parallel Prolog system was developed to serve as a vehicle for investigating primitives by which to extend Prolog for the purpose of parallel (object oriented) processing.

Each of the examples presented, motivated the choice for a particular construct.

The communication construct, involving channels, was directly inspired by my previous experience in developing a semantics for Occam. A notable difference with the construct of Occam however is that, instead of value assignment, a unification between the input term and the output term must occur in both processes that take part in the communication. My first choice was for a symmetric construct that forced both processes to backtrack when the communication failed. The example of a *counter*, nevertheless, indicated that an asymmetric construct, allowing the input side to backtrack until a unifying input statement occurs, is a more natural choice. This solution slightly complicated the communication protocol, since the output side must be notified when the input side succeeds in finding a proper input statement.

As it turns out, communication over channels seems to be no longer necessary, once classes are provided with the capability to store values in non-logical variables and to postpone the evaluation of a goal until a certain condition is satisfied. Nevertheless, from a semantic point of view, these extensions are rather complex since they necessitate to give an account of the interplay between local states of instances of classes and the derivations performed by processes referring to it. In contrast, adding channels seems to be a relatively simple extension to sequential Prolog.

The variety of examples shows that a parallel object oriented Prolog as proposed allows to solve many of the problems found in parallel programming. More examples are needed however to investigate the full potentiality of a language of this kind.

Moreover, a number of questions need to be answered to get a clear insight in the particularities of such a language. Notably the relations between backtracking, communication, and synchronisation by means of answer statements must be studied. Another problem is how the cut operator affects the behavior of a collection of processes. Cf. [Ko88], [dB88].

Having developed the language thus far, it seems a good moment to hand over these questions to the specialists in the field, to clarify the problems involved.

Acknowledgements
I wish to thank Carel van den Berg, who developed some of the tools without which I would never

have undertaken this effort, and who moreover showed an early interest in this project. Also I would like to thank Joost Kok, for suggesting the name POOLOG, and for showing an interest in working at a semantics for this language. Pierre America is thanked for allowing me to be inspired by his language, and for reading a first draft of this manuscript.

References

[Al86] L. Allison *A practical introduction to denotational semantics* Cambridge Computer Science Texts 23, 1986

[Am87] *POOL-T: a parallel object oriented language* in: A. Yonezawa and M. Tokoro (eds.) Object oriented concurrent systems MIT Press 1987

[Au85] L. Augusteijn *POOL-T User Manual* Report Esprit project 415 Doc Nr 0104 Philips Research Laboratory Eindhoven

[dB88] J.W. de Bakker *Comparative semantics for flow of control in logic programming without logic* Report CS-R8840, CWI

[BL86] R. Butler, E. Lusk, W. McCune and R. Overbeek *Parallel logic programming for numeric applications* LNCS 225, 1986, pp. 375-388

[Bo89] *Parallelism in Logic Programming* K. De Bosschere Report DG 89-02 Lab. voor Electronica en Meettechniek, Gent, 1989

[Co85] J. Cohen *Describing Prolog by its interpretation and compilation* CACM, Dec 1985

[Co87] J.S. Conery *Parallel execution of logic programs* Kluwer, Boston 1987

[HA84] R. Hasegawa and M. Amamiya *Parallel execution of logic programs based on dataflow concept* Proc. FGCS 1984, ICOT, pp. 507-516

[HN86] M.V. Hermenegildo and R. Nasr *Efficient management of backtracking in and-parallelism* LNCS 225, 1986, pp. 40-55

[DG84] D. DeGroot *Restricted and-parallelism* Proc. FGCS 1984, ICOT, pp. 471-478

[In84] Inmos LtD *The Occam Programming Manual* Prentice Hall International London 1984

[Ko88] J.N. Kok *A compositional semantics for Concurrent Prolog* Proc. STACS, Bordeaux 1988

[PM85] L. Pereira, L. Monteiro, J. Cunha and J. Aparico *Delta Prolog: a distributed backtracking extension with events* LNCS 225, 1986, pp. 69-83

[PN84] L.M. Pereira and R. Nasr *A distributed Logic programming Language* Proc. FGCS 1884, ICOT, pp. 283-291

[Ri88] G.A. Ringwood *Parlog86 and the dining logicians* Comm. ACM Vol. 31 No. 1 pp. 10-25 1988

[ST83] E. Shapiro, A. Takeuchi *Object-oriented programming in Concurrent Prolog* New Generation Computing, Vol. 1, no. 2 1983