PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1984,
edited by ICOT. © ICOT, 1984

NOTES ON SYSTEMS PROGRAMMING IN PARLOG

Keith Clark and Steve Gregory

Department of Computing, Imperial College
London SW7 2BZ, England

ABSTRACT

Several topics connected with systems programming in the parallel logic programming language PARLOG are discussed.

We argue that a parallel language needs a much more elaborate metacall facility than the simple succeed-fail metacall of PROLOG. In order to program an operating system shell which is failsafe, allows abort termination of processes and which makes visible any incrementally constructed output of a user process, a three-argument metacall primitive is needed which always succeeds. The first argument is the call to be evaluated, the second is the status or result of the evaluation and the third is an input argument which can be used by some other process to control the evaluation of the call.

1 INTRODUCTION

1.1 Brief introduction to PARLOG

PARLOG (Clark and Gregory 1984a) is a parallel logic programming language featuring both and- and or-parallelism. For this paper we need to use only the and-parallel subset of PARLOG, which we shall briefly outline here. This language, based on Horn clauses, differs from PROLOG in three crucial respects: "don't care non-determinism", parallel evaluation and "mode" declarations to specify communication constraints on shared variables. Each relation call can be evaluated as a separate process. The shared variables act as communication channels along which messages are sent by incremental construction of streams, which are lists of message terms.

1.1.1 Don't care non-determinism

A PARLOG clause takes the form

```
r(t1,...,tk) <- <guard conditions> :
                <body conditions>.
```

where the : signals the end of the guard and **t1,...,tk** are argument terms.

Both the **<guard conditions>** and the **<body conditions>** are conjunctions of relation calls. There are two types of conjunction: the parallel "and" **(C1 , C2)** in which the conjuncts **C1** and **C2** will be evaluated in parallel, and the sequential "and" **(C1 & C2)** where **C2** will only be evaluated when **C1** has successfully terminated.

In the evaluation of a relation call **r(t1',...,tk')**, all of the clauses for relation **r** will be searched in parallel for a candidate clause. The above clause is a candidate clause if the head **r(t1,...,tk)** matches the call **r(t1',...,tk')** and the guard succeeds. It is a non-candidate if the match fails or the match succeeds and the guard fails. If all clauses are non-candidates the call fails, otherwise one of the candidates is selected and the call is reduced to the substitution instance of its body. There is no backtracking on the choice of candidate clause. We "don't care" which candidate clause is selected. In practice, the first one (chronologically) to be found is chosen.

During the search for a candidate clause, no variables in the call are bound. There is no output binding to variables of the call until the evaluation commits to the use of some clause. Because there is no backtracking there is never any need to rescind a message sent via a shared variable of the call.

The search for a candidate clause can be controlled by the use of the ; operator between clauses. If a relation is defined by the sequence of clauses

```
Clause1.
Clause2;
Clause3.
```

Clause3 will not be tried for candidacy until both **Clause1** and **Clause2** have been found to be non-candidate clauses.

1.1.2 Modes

For every PARLOG relation definition there is a mode declaration which states whether each argument is input (?) or output (^). For example, the relation **merge(x,y,z)** has the mode **(?,?,^)** to merge lists **x** and **y** to list **z** (lower case identifiers are variables):

```
mode merge(?,?,^).
merge([u|x],y,[u|z]) <- merge(x,y,z).
merge(x,[v|y],[v|z]) <- merge(x,y,z).
merge([],y,y).
merge(x,[],x).
```

Concurrently evaluating relation calls communicate via shared variables; the modes impose a direction on this communication. Non-variable terms that appear in input argument positions in the head of a clause can only be used for input matching. If an argument of the call is not sufficiently instantiated for an input match to proceed, the attempt to use the clause suspends until some other process further instantiates the input argument of the call. For example, the first clause for **merge** has **[u|x]** in its first input argument position. Until the call has a list or partial list structure of the form **[u|x]** in the first argument position the first clause is suspended.

If all clauses for a call are suspended, the call suspends. A candidate clause can be selected even if there are other, suspended, clauses.

1.2 Concurrent PROLOG

Concurrent PROLOG (hereafter CP) (Shapiro 1983) is very similar to the and-parallel subset of PARLOG; the main difference is that CP uses read-only variable annotations on variables to specify the communication constraints, where PARLOG uses modes.

Because both PARLOG and CP are derivatives of the Relational Language of (Clark and Gregory 1981), they both feature don't care non-determinism, guarded clauses and the property that no bindings are made to a call until the evaluation commits to the use of some clause. (However, Shapiro has recently proposed a possible relaxation of this last property; we discuss this in section 8.)

1.3 Systems programming in PARLOG

In (Shapiro 1984), Shapiro develops in CP a failsafe Unix-like shell program to run foreground and background commands and handle **ABORT** interrupts for foreground commands. His approach relies on evaluating commands as conventional success-or-fail metacalls in the guards of the shell program. The fundamental problem with this approach is that the output of guard processes is not made available until the guard terminates. Hence, incrementally constructed output of a user process is not visible whilst the process is evaluating. This prevents interactive communication between a user process and the user or between a user process and a filestore process, for example.

We follow through the examples of Shapiro's paper and show how they can be rewritten with user commands evaluated outside the guards, and hence with incrementally constructed data made visible, by using a three-argument metacall primitive that always succeeds. The first argument is the call to be evaluated, the second is the status or result of the evaluation (e.g. **SUCCEEDED, FAILED**) and the third is an input argument which can be used by some other process to control the evaluation of the call.

The use of this primitive not only allows interactive user programs but allows us to program a more powerful shell that allows the selective aborting of background processes as well as foreground processes. It also permits scheduling strategies to be imposed on processes.

As well as solving several problems in systems programming, the three-argument metacall subsumes certain PARLOG features such as negation as failure and the sequential "and", and allows us to reduce the or-parallel evaluation of the guards of alternative clauses to and-parallel evaluation. We contend that for these reasons the three-argument metacall is the natural metacall primitive for parallel logic programming languages like PARLOG and CP.

2 A SIMPLE SHELL

We begin by writing in PARLOG (Program 1) a simple shell that handles a stream of commands to run foreground and background processes without input or output. The relation **shell(cmds)** acts as a process which consumes a stream of commands **cmds** and invokes each as a process using the **call** metacall. The commands are labelled by **FG** (foreground) or **BG** (background).

```
mode shell(?).
shell([]).                                   (S1)
shell([BG(proc)|cmds]) <-
   call(proc), shell(cmds).                  (S2)
shell([FG(proc)|cmds]) <-
   call(proc) & shell(cmds).                 (S3)
```

Program 1: a simple shell

Clause (S1) terminates the shell when the command list is closed. (S2) deals with a background command **BG(proc)** by invoking **proc** concurrently with resuming the shell to process the next command. (S3) is similar but handles foreground commands. It waits for the command process to terminate successfully before accepting the next command. This is due to the use of the sequential "and" (&) in place of the parallel "and" (,).

Program 1 is similar to the one given in (Shapiro 1984) except that, in the latter, foreground commands are evaluated in a guard as in clause (SS3).

```
   shell([FG(proc)|cmds]) <-
      call(proc) : shell(cmds).               (SS3)
```

The use of the guard to enforce sequential execution is acceptable only if the foreground command does not produce any output, or if it is acceptable that all the output generated by the foreground process is only visible to the **shell** user when the process terminates. More realistically, the **shell** process should explicitly merge output streams from each invoked process so that messages from the processes can be displayed to the user or transmitted to other processes in the operating system.

3 A SHELL WITH OUTPUT

To make our shell program more useful we allow commands to produce stream output which is accessible as stream output of the **shell** whilst the process is running. This is implemented by Program 2, for the relation **shell(cmds,so)**. **cmds** is an input list of commands, as before except that a command is now of the form **BG(proc,co)** or **FG(proc,co)**. **proc** is the process to be executed while **co** is the output stream of **proc** that should be passed out of the **shell** process via the **shell** output stream **so**. For example, a possible command is **BG(primes(x),x)** where **x** is the stream of primes to be displayed at the terminal.

```
   mode shell(?,^).
   shell([],[]).                              (OS1)
   shell([BG(proc,co)|cmds],so) <-
      merge(co,nso,so),
      call(proc), shell(cmds,nso).            (OS2)
   shell([FG(proc,co)|cmds],so) <-
      merge(co,nso,so),
      ( call(proc) & shell(cmds,nso) ).       (OS3)
```

Program 2: a shell with output from commands

Each time a command is received, the shell creates a new **merge** process to run concurrently with the command. (The **merge** in (OS3) could be replaced by **append** with no change to the behaviour.) Any output generated by the command process is merged onto the shell's output stream immediately. This is true of both foreground and background commands. If we were to follow Shapiro's method of placing the call to **proc** in a guard in (OS3), any output generated by a foreground command would be invisible until the command terminated. This would make it impossible to run interactive foreground programs.

This example illustrates our point that the guard should not be used to enforce sequential evaluation; it is far too powerful. What is needed is the sequential **P & Q** construct which delays the evaluation of **Q** without delaying the output of **P**.

By running **shell** in conjunction with a message handler we can allow any sort of output message to be produced by a process, including requests for input via variables in the messages. Thus, a message of the form

```
   filestore(GET,filename,x)
```

would be routed by the message handler as a **GET** request to the filestore. The retrieved file would be returned to the requesting process as the binding for the variable **x** which will be local to the sending process. In this way the processes being run by the **shell** can have input as well as output communication. For more details of the technique of two-way communication using variables in messages, which is due to Shapiro, we refer the reader to (Shapiro 1983) or to (Clark and Gregory 1984a).

4 PROCESS FAILURE

As Shapiro points out in (Shapiro 1984), the shell of Program 1 will crash if any of the commands fails since they are part of the same conjunction as the recursive shell invocation. To overcome this, he replaces the metacall **call(proc)** by **envelope(proc)** which always succeeds. The definition of **envelope** can be adapted to PARLOG as follows:

```
   mode envelope(?).
   envelope(proc) <- call(proc) :;
   envelope(proc).
```

where we have used the sequential search operator ; between the clauses to ensure that the second clause is used only if the guard of the first clause fails.

This suffers from the same fatal flaw as that described in the previous section: since the command is evaluated in a guard its output will not be made available until it has successfully terminated.

4.1 A two-argument metacall

Our solution to this problem is to generalize the metacall primitive by adding a second argument:

```
   call(goal?,status^)
```

Such a metacall evaluates its first argument **goal** and always succeeds with an output binding for **status**: **SUCCEEDED** or **FAILED** depending on the success or failure of **goal**. Any output generated by the evaluation of **goal** is available immediately, as it would be in a call **call(goal)**. The difference is that even if **goal** subsequently fails, the output generated up to the point of failure remains since the metacall succeeds.

The behaviour of a shell which evaluates commands (i.e. user programs) using this more general metacall seems to be what one would expect of a practical operating system. If a user program fails (resulting in a **FAILED** result from the metacall), the operating system will not crash. Moreover, output from a user program is incrementally available whether or not it ultimately fails. The output trace of a failed program is therefore available for debugging and other purposes.

A third possible value of **status** is **ERROR**, which will be issued if a run-time error occurs during the evaluation of a metacall. Again, the metacall itself will succeed in this case. We can now define a relation **terminated(status)** which succeeds when a metacall evaluation has finished:

```
mode terminated(?).
terminated(SUCCEEDED).
terminated(FAILED).
terminated(ERROR).
```

More generally, the **ERROR** message might be parameterized to include information about the type of error (invalid use of some primitive for example) and the call that resulted in the error. Finally, if the metacall is further generalized to accept inputs via bindings for variables in its error messages, we have a building block with which to implement error recovery.

4.2 Sealing the output stream of a terminated process

A program that fails or encounters an error (or even succeeds) before it has finished its output will leave a "dangling stream", i.e. a list with a variable as some tail sublist. The consumer of the output of a program evaluated by the two-argument metacall must therefore monitor the result of the program and close the dangling stream if the program terminates. For example, the second clause of Program 2 must be changed to

```
shell([BG(proc,co)|cmds],so) <-
   dmerge(status,co,nso,so),
   call(proc,status),
   shell(cmds,nso).                    (OS2')
```

Program 3 defines **dmerge** which is similar to **merge** except that it has an extra argument: the **status** of the metacall process producing its first input stream. The extra fifth clause effectively closes this stream on the termination of its producer process by terminating the **dmerge** process when its **s** argument is **SUCCEEDED**, **FAILED** or **ERROR** and there is no output from the metacall waiting to be passed through (the **var** test). The **s** argument is acting as a termination message about the first input stream.

```
mode dmerge(?,?,?,^).
dmerge(s,[u|x],y,[u|z]) <- dmerge(s,x,y,z).
dmerge(s,x,[v|y],[v|z]) <- dmerge(s,x,y,z).
dmerge(s,[],y,y).
dmerge(s,x,[],x).
dmerge(s,x,y,y) <- terminated(s), var(x) :.
```

Program 3: a merge with a termination message

5 ABORTING PROCESSES

In (Shapiro 1984) Shapiro extends his shell program so that the current foreground process is aborted on receipt of an **ABORT** (or Control-C) interrupt on the command stream. His solution again relies on the execution of the command process in the shell's guard as in (SS3). He does this by having two clauses to handle foreground commands:

```
shell([FG(proc)|cmds]) <-
   call(proc) : shell(cmds).           (SS3)
shell([FG(proc)|cmds]) <-
   search(ABORT,cmds,ncmds) :
   shell(ncmds).                       (SS4)

mode search(?,?,^).
search(u,[u|x],x).
search(u,[v|x],y) <- u =/= v :
   search(u,x,y).
```

The command process of the metacall **call(proc)** runs in parallel with the process searching for an **ABORT** since both are in the guards of clauses and will be evaluated in parallel in the search for a candidate clause. The successful termination of either guard process aborts the other guard and the **shell** process is continued at the appropriate point in the input command stream.

For the reasons that we have already given regarding the need to access output of commands during their evaluation, we regard any solution that places a command evaluation in a guard as unsatisfactory. Moreover, the racing of guards will only allow abort termination of foreground processes, or of all processes.

5.1 A general metacall primitive

Our solution to the problem is once more to generalize the metacall to a three-argument form:

```
call(goal?,status^,control?)
```

The third argument **control** will normally be an uninstantiated variable. If it is bound to the term STOP by another process, the evaluation of **goal** will be terminated with **status** bound to STOPPED. We must now add another clause to our **terminated** relation:

```
terminated(STOPPED).
```

5.2 Aborting foreground commands

Program 4 gives our version of a shell that handles **ABORT** interrupts for foreground processes with an **ABORT** command given in the normal command stream. When a foreground command is received (AS3), the evaluation of the command and the search for an **ABORT** command are invoked as and-parallel metacalls, each of which can be prematurely terminated by a **STOP** message from the third process, **arb**.

The **arb** process monitors the results **s1** and **s2** of these metacalls: it **STOP**s the command process if the search for an **ABORT** is successful, or **STOP**s the search if the command process terminates. It also selects the appropriate command stream continuation point (**cmds** or **acmds**) depending upon whether the user process or the **ABORT**-seeking process has terminated first. This is then passed on to the recursive **shell** process which starts as soon as the **arb** process has terminated, either on a normal termination of the foreground process or on an **ABORT** being found.

```
mode shell(?), arb(?,?,^,^,?,?,^).

shell([]).                                    (AS1)
shell([BG(proc)|cmds]) <-
   call(proc,s,c), shell(cmds).               (AS2)
shell([FG(proc)|cmds]) <-
   call(proc,s1,c1),
   call(search(ABORT,cmds,acmds),s2,c2),
   ( arb(s1,s2,c1,c2,cmds,acmds,ncmds) &
     shell(ncmds) ).                          (AS3)

arb(s1,s2,c1,STOP,cmds,acmds,cmds) <-
   terminated(s1) :.
arb(s1,SUCCEEDED,STOP,c2,cmds,acmds,acmds).
```

Program 4: a shell that handles ABORT interrupts for foreground processes

5.3 Aborting background commands

Program 5 has an output stream for commands (as in Program 2) and allows the aborting of background commands. It does this by keeping a **procs** list of all the active background processes (those whose result variables have not yet been instantiated) identified by a **proc-id**. A message linking the command with its **proc-id** is output to the user when the process is invoked. When the special command **KILL(proc-id)** is received, the current **procs** list is searched by the **kill** process and the identified process is stopped by setting its **control** argument to **STOP**, using the assignment unification primitive :=. The **insert** process that adds a new record to the process list also generates the **proc-id** and may also garbage collect the current process list by deleting all the processes with a bound **status** variable.

The clause for **kshell** that deals with foreground commands has been omitted. It will be a slight modification of that in Program 4 to allow for an output stream merge. It need not add the foreground process to the process list and it can abort as before on an **ABORT** command.

```
mode shell(?,^), kshell(?,?,^),
     kill(?,?,^).

shell(cmds,so) <- kshell(cmds,[],so).

kshell([],procs,[]).                          (KS1)
kshell([KILL(proc-id)|cmds],procs,
       [KILLED(proc-id)|so]) <-
   kill(proc-id,procs,nprocs),
   kshell(cmds,nprocs,so).                    (KS2)
kshell([BG(proc,co)|cmds],procs,
       [NEW-PROC(proc-id,proc,co)|so]) <-
   insert(PROC(proc-id,s,c),procs,nprocs),
   dmerge(s,co,nso,so),
   call(proc,s,c),
   kshell(cmds,nprocs,nso).                   (KS3)

kill(proc-id,[],[]).
kill(proc-id,[PROC(proc-id,s,c)|procs],
             procs) <-
   c := STOP.
kill(proc-id,[PROC(p-id,s,c)|procs],
             [PROC(p-id,s,c)|nprocs]) <-
   proc-id =/= p-id :
   kill(proc-id,procs,nprocs).
```

Program 5: a shell that handles KILL commands for background processes

6 PRIORITY SCHEDULING

6.1 Suspending evaluations

Our final refinement to the general metacall primitive **call(?,^,?)** is to allow the evaluation of a metacall to be temporarily suspended and restarted by another process. The **control** argument can now be bound incrementally to a list of **SUSPEND** or **CONTINUE** messages, possibly terminated by the term **STOP**. Each time a **SUSPEND** message is sent on the **control** argument, the message is echoed on the **status** argument and the evaluation enters a suspended state. It can only be resumed by sending a **CONTINUE** message, which again is echoed on **status**.

6.2 A priority shell

In our previous shell programs, background processes continue running even when a foreground process is invoked. We might wish to give a higher priority to the foreground process, so that background processes run only when there is no active foreground process. This is implemented by Program 6.

```
mode shell(?), pri-shell(?,?).

shell(cmds) <- pri-shell(cmds,bgc).

pri-shell([],bgc).                              (PS1)
pri-shell([BG(proc)|cmds],bgc) <-
   call(proc,s,bgc),
   pri-shell(cmds,bgc).                         (PS2)
pri-shell([FG(proc)|cmds],bgc) <-
   bgc := [SUSPEND|bgc1] &
   call(proc,s,fgc) &
   bgc1 := [CONTINUE|nbgc] &
   pri-shell(cmds,nbgc).                        (PS3)
```

Program 6: a shell with priority to foreground commands

Background commands are evaluated in parallel with the shell, as before, but share a common **control** argument. When a foreground command is invoked, a **SUSPEND** message is sent on this **control** argument, causing all background processes to suspend. When the foreground command terminates, a **CONTINUE** is sent, reactivating the background processes.

7 A PRIORITY SHELL WITH INPUT

We now treat the case of a foreground command taking input data from the shell's command stream. This input must be demand-driven: the command process will generate a stream of request variables, each of which will be bound to the next data item on the command stream when it is available. A command of the form **FG(proc,ci)** will invoke **proc** as a foreground process and treat **ci** as a list of variables to demand items from **cmds**. When the foreground command terminates, the remainder of **cmds** will be passed back to the shell.

As in Program 6, we shall give a foreground process priority over background processes. However, if a foreground process has to wait for input from the command stream, control can be relinquished to the background processes. This is implemented by Program 7, which gives a new fourth clause (PS4) to be added to the shell program of Program 6.

```
mode switch(?,?,^,^,?,?,^),
     sw(^,?,?,^,^,?,?,^).

pri-shell([FG(proc,ci)|cmds],bgc) <-
   bgc := [SUSPEND|bgc1],
   call(proc,s,fgc),
   switch(ci,s,fgc,bgc1,nbgc,cmds,ncmds),
   pri-shell(ncmds,nbgc).                     (PS4)

switch(ci,s,fgc,[CONTINUE|bgc],bgc,
      cmds,cmds) <- terminated(s) :.
switch([req|ci],s,fgc,bgc,nbgc,
      [data|cmds],ncmds) <- var(s) :
   req := data,
   switch(ci,s,fgc,bgc,nbgc,cmds,ncmds).
switch([req|ci],s,[SUSPEND|fgc],
      [CONTINUE|bgc],nbgc,cmds,ncmds) <-
   var(s), var(cmds) :
   [SUSPEND|ns] <= s,
   sw(req,ci,ns,fgc,bgc,nbgc,cmds,ncmds).

sw(req,ci,s,fgc,bgc,bgc,cmds,cmds) <-
   terminated(s) :.
sw(data,ci,s,[CONTINUE|fgc],[SUSPEND|bgc],
   nbgc,[data|cmds],ncmds) <- var(s) :
   [CONTINUE|ns] <= s,
   switch(ci,ns,fgc,bgc,nbgc,cmds,ncmds).
```

Program 7: extensions to Program 6 to allow input

When a foreground command of the form **FG(proc,ci)** is received, a **SUSPEND** message is sent to the background processes, as in (PS3), and the command process is evaluated. A **switch** process monitors the status **s** of the evaluation and the stream of request variables **ci** that it generates.

There are three cases for **switch**. The first clause handles the termination of the foreground process: it sends a **CONTINUE** message to the background processes and passes back to the shell the current point in the (input) command stream and the (output) background process control stream. The second clause is a candidate if the foreground process has not terminated (the **var(s)** test in the guard) and there is a request for input on **ci** and data is available on the command stream (e.g. by type-ahead). In this case, the available data is assigned to the request variable.

The third clause for **switch** applies when the foreground process has not terminated and there is a request for input but no data is available on the command stream (the **var(cmds)** test). Now a **SUSPEND** message is sent to the foreground process, a **CONTINUE** sent to the background processes and the **switch** process enters a new state **switch1**. The call to <=, the PARLOG matching unification primitive, skips over the **SUSPEND** message that is echoed on the status stream of the foreground process.

While the foreground process is waiting for input there are two possibilities, handled by **switch1**. The first clause handles the termination of the process as before. The second clause is applicable if an item arrives on the shell's command stream. In this case, the item is assigned to the request variable, the background processes are suspended and the foreground process is resumed.

8 CONCLUDING REMARKS

8.1 Discussion

We have proposed the addition of a new metacall primitive **call(?,^,?)** into PARLOG and related languages to facilitate the writing of operating systems. The one-argument and two-argument forms of **call** can be defined in terms of this primitive:

```
mode call(?,^).
call(g,s) <- call(g,s,c).

mode call(?).
call(g) <- call(g,s,c), s = SUCCEEDED.
```

The metacall approach seems sound since operating systems are necessarily at a different level than user programs. An operating system program is not concerned with the details of user programs, only the results that they produce. In addition the system must be able to initiate, terminate and suspend the execution of user programs. The proposed primitive meets these criteria.

The suitability of the three-argument metacall as the metacall primitive is reinforced by the fact that it can be used to program two other control features of PARLOG: negation and the sequential "and", and to reduce the or-parallel evaluation of alternative guards to an and-parallel evaluation.

Program 8 defines negation as failure, and **&**, using the two-argument form of **call**. It evaluates **a & b** by executing a call to a in parallel with a process which is input-suspended until the evaluation of a succeeds, whereupon b is called.

```
mode ~ ?.
~ a <- call(a,s), s = FAILED.

mode ? & ?.
a & b <- call(a,s), nextcall(s,b).

mode nextcall(?,?).
nextcall(SUCCEEDED,b) <-
   call(b).
```

Program 8: definition of negation and sequential "and"

PARLOG programs with clauses with non-empty guards can also be compiled into programs with a single guard clause in which the different guards are evaluated in and-parallel using three-argument metacalls in a manner similar to that used in Program 4. An arbitration process then monitors the results of these guard metacalls, selects the appropriate body, and explicitly kills the other guard metacalls. The details of this representation are given in (Clark and Gregory 1984b).

The general metacall primitive should be readily implementable on any architecture that properly supports PARLOG or similar languages. This is certainly true of ALICE (Darlington and Reeve 1981). All that is required is the ability to access the result of an evaluation (**SUCCEEDED**, **FAILED**, **ERROR**) and to terminate and suspend an evaluation.

The primitive has been implemented in the PROLOG-based PARLOG system (Gregory 1984). Copies of this system, for either micro-PROLOG or DEC-10 PROLOG, are available from the authors. It has been used to test all example programs presented in this paper.

8.2 Related work

"Write-early" variables have been proposed recently by Shapiro and described in (Furukawa et al. 1984) as a way of making the output of a guard visible before the guard has succeeded. A write-early annotation ^ placed on a variable in a goal signals that any bindings to the variable are made public immediately, even if they occur in clause guards. This feature can be used to partially implement the three-argument **call** as follows (CP syntax is used):

```
call(Goal,Status,Control) :-
   call1(Goal^,Status,Control?).
```

```
call1(Goal,stopped,stop).
call1(Goal,Status,Control) :-
   call(Goal,Status) | true.

call(Goal,succeeded) :- call(Goal) | true.
call(Goal,failed) :- otherwise | true.
```

The use of the write-early annotation is potentially very dangerous: it overrides the security of the elaborate and expensive unification of CP. This approach is an attempt to use the guard for purposes for which it is not intended. Our approach is to design the three-argument **call** as the primitive (it is a simple and flexible concept), and then use it to implement the guard as can be done in PARLOG.

The three-argument **call** has some similarities to the meta predicate **simulate** proposed for KL1 (Furukawa et al. 1984):

```
simulate(World,NewWorld,Goal,Result,Control)
```

This evaluates **Goal** relative to a local program **World** which can be updated to **NewWorld.** The **Control** argument here specifies some control strategy to be used, such as breadth-first or depth-first, while **Result** returns arbitrarily detailed information about the progress of the evaluation.

Like **call, simulate** can be used to obtain the success/failure result of an evaluation. It is not intended to allow an evaluation to be stopped or suspended, though there should be no difficulty in allowing this. As it is described in (Furukawa et al. 1984), it appears that an important difference between **call** and **simulate** is that the latter does not allow incremental communication between an evaluation ("object world") and the outside program ("meta world") unless write-early variables are used.

simulate is intended as a general purpose meta inference predicate having many different uses, some of which seem to require an interpretive evaluation. In contrast, we are proposing the three-argument **call** as a primitive of the language: one which is simple enough to implement efficiently but powerful enough to enable the programming of realistic operating systems.

ACKNOWLEDGEMENTS

This work was supported by the Science and Engineering Research Council under grant number GR/B/97473.

Many of the ideas germinated in discussions with Udi Shapiro and the members of the KL1 design team whilst the authors were visiting ICOT in October 1983. We would like to thank ICOT for inviting us and for providing a very stimulating research environment.

REFERENCES

Clark K.L. and Gregory S. (1981), A relational language for parallel programming. In _Proc. Conf. on Functional Programming Languages and Computer Architecture_, ACM, pp 171-178.

Clark K.L. and Gregory S. (1984a), PARLOG: parallel programming in logic. Research report DOC 84/4, Dept. of Computing, Imperial College, London.

Clark K.L. and Gregory S. (1984b), Notes on the implementation of PARLOG. Research report, Dept. of Computing, Imperial College, London.

Darlington J. and Reeve M.J. (1981), ALICE: a multi-processor reduction machine. In _Proc. Conf. on Functional Programming Languages and Computer Architecture_, ACM, pp 65-75.

Furukawa K., Kunifuji S., Takeuchi A. and Ueda K. (1984), The conceptual specification of the Kernel Language version 1. Technical report, ICOT, Tokyo.

Gregory S. (1984), How to use PARLOG. Unpublished report, Dept. of Computing, Imperial College, London.

Johnson S.D. (1981), Circuits and systems: implementing communications with streams. Technical report 116, Dept. of Computer Science, Indiana University.

Shapiro E.Y. (1983), A subset of Concurrent Prolog and its interpreter. Technical report TR-003, ICOT, Tokyo.

Shapiro E.Y. (1984), Systems programming in Concurrent Prolog. In _Proc. 11th ACM Symp. on Principles of Programming Languages_.