

Localising the GHC suspension test

Hamish Taylor

Department of Computer Science
Heriot-Watt University, Edinburgh, U.K.

Abstract

Existing proposals for implementing the run-time suspension test for GHC concentrate upon suspending unifications in the bodies of clauses being used to evaluate guard literals. However, by making simple alterations to the guard and adding special primitives to it to localise the requirement to suspend to a primitive in the guard, unifications in the bodies of clauses may proceed without being subject to any suspension check. The result is a clause to clause translation technique from full GHC to a safe variant language, the parallel subset of kernel Parlog, where no unification in the body of a clause suspends. Emulating a translated clause in this language suffices to emulate the original GHC clause. This technique simplifies the implementation of full GHC and allows Parlog implementations with moderate extensions to include a run-time guard safety check where compile-time safety analysis is problematic to apply correctly.

Keywords GHC, Parlog, suspension, guard safety, parallel execution

1. Introduction

Execution of GHC is governed by a rule of synchronisation and a rule of sequencing [1]. The rule of synchronisation states that the attempt to bind a calling argument variable in the guard prior to commitment should cause the unification to suspend. The rule of sequencing states that the attempt to bind a guard variable during execution of the body prior to commitment should also cause the unification to suspend. The latter suspension test is only needed in an implementation that allows the execution of the bodies of clauses before commitment. A lot of redundant execution could result from executing the bodies of GHC clauses that will never be trusted. At best it would only be worthwhile doing on highly parallel architectures.

Saraswat argues that GHC clauses, whose bodies may be executed before their head arguments and guards are satisfied, introduce semantic problems into the language anyway [2]. For these reasons the focus will only be on the rule of synchronisation. The assumption will be that no part of the body will be executed before the clause has committed. However what applies to the rule of synchronisation could also be applied to the rule of sequencing.

Two of the main approaches canvassed for implementing the guard suspension test are :

- 1) Ueda's Pointer Colouring Scheme [1, 3]
- 2) Miyazaki's Guard Numbering Scheme [4]

Both schemes concentrate upon suspending unifications in the bodies at GHC clauses being used to evaluate guard literals in the event of the unification attempting to write upon a calling argument variable.

Either scheme threatens to impose a suspension test overhead upon every explicit unification. However, most GHC clauses either only input match upon head arguments or have only system primitives with input arguments for guards. Only in evaluating the remaining cases is it necessary to test for a user defined guard literal or system primitive with output arguments writing upon a calling argument variable. A more demand driven scheme would separate out the suspension producing conditions from GHC clauses, in order to localise the requirement to suspend to evaluation of an extra system literal in the responsible guard, and so impose no general overhead upon unification in the body. By allowing unsafe guard literals to be evaluated without suspension constraints on their output bindings, evaluation is made more eager and computation space is more effectively recycled. Computation space used in evaluating the guard literal is claimed and released as soon as the guard literal can be satisfied or fails. It is not claimed and then frozen unused pending the further communication of input values to unsuspend a suspended unification. For deep guard evaluations this frozen space may include space claimed to store ancestor processes suspended upon the unification process as well as that claimed to store the unification process.

Translating away the run-time suspension test of GHC allows an emulator of a safe variant language to execute full GHC by emulating clauses translated into that language instead. As will be seen later, this safe variant language needs to contain extra system primitives that cannot be defined in GHC, so it is not correct to call the safe variant language "safe GHC", the subset of GHC where

there are no suspensions of body unification goals in all possible computations. It is more apt to call the safe variant language the parallel subset of kernel Parlog [5]. Kernel Parlog is not defined in a way that constrains it from accepting system primitive extensions. Kernel Parlog is the result of translating away mode declarations and head argument matches from Parlog.

2. Committed Choice Languages

The two other main committed choice parallel logic programming languages currently being investigated are Concurrent Prolog [6, 7] and Parlog [5]. Each language has its respective advantages [8] and their respective relationships with GHC are becoming clearer.

Levy and Shapiro [9] have described a way in which safe GHC can be compiled into Safe Concurrent Prolog which in turn can be compiled into Flat Concurrent Prolog or FCP. More recently Saraswat has proposed a way of translating GHC into a variant of Concurrent Prolog he calls $CP(\downarrow, |)$ [2]. Saraswat argues a similar translation establishes flat GHC/Parlog is a subset of FCP. Unlike the method of translation proposed here, Saraswat's scheme of translation is not directed at the issue of implementing GHC. His mapping of GHC clauses to $CP(\downarrow, |)$ clauses so as to preserve success sets is too complex and unwieldy to be practical. Saraswat is only interested in arguing through his translation scheme that $CP(\downarrow, |)$ subsumes GHC in expressive power. However in the absence of a serious implementation of $CP(\downarrow, |)$, this theoretical result lacks practical significance.

Flat GHC and the parallel subset of flat kernel Parlog are essentially equivalent languages in expressive power. The main differences between non-flat GHC and Parlog are that guard safety is implemented in Parlog by a compile-time safety check and in GHC by a run-time suspension test and that GHC eschews meta-calls and primitives like *var/2* that have time of call semantics. GHC can express a succinct meta-interpreter of itself in itself and Parlog can interpret itself using its meta-call facility. With the method of translation of GHC to the parallel subset of kernel Parlog proposed in this paper, GHC should be executable on any implementation of kernel Parlog that has been extended by suitable system primitives. Since kernel Parlog is an intermediate but simpler representation of Parlog in the established process of compiling Parlog, this really means that GHC should be executable on any suitably extended Parlog implementation.

Gregory observes in relation to Parlog about guard safety checks that are complex or impossible that

"It may be possible to find a way to combine the efficiency advantages of a compile-time safety check with the power of a run-time test where this is required." (p.132 [51])

The method of translation described in this paper provides the means to augment a compile time safety check on a Parlog implementation with a run time suspension test. It also allows suitably extended implementations of a language underlying GHC and Parlog to combine the declarative cleanness of GHC with the procedural control of Parlog on a single implementation. Parlog clauses for certain relations whose guard safety is unproblematic to determine automatically and GHC clauses for others, where this requirement does -not hold, can be translated into a lingua franca for emulating both that will ensure guard safety for all unproblematically.

3. Localisation

The central idea behind localisation is that suspension effects due to the rule of synchronisation arise from what happens during guard evaluation and can be restricted to the locus of that evaluation. In what follows variables will be termed unsafe that occur both in the heads of GHC clauses and either in user defined guard literals or argument positions of system guard literals liable to bind given values. If all unsafe variables in guard literals are replaced by fresh variables then all transformed calls to guard literals can be allowed to proceed without being subject to a safety suspension condition. Each safety suspension condition can then be achieved by a special purpose system predicate that relinks the unsafe guard variable with its new replacement. In this way a GHC program can be distilled down to guarded Horn clauses interleaved with special primitives to achieve the safe guard suspension mechanism.

The function of the special primitive will be to allow the passing of values from the old variable to the new but to ensure that if unifying the new and the old guard variables would bind a calling argument variable before commitment, this special primitive will suspend. As an example take the GHC clause

```
(1) head([A]) :- test(A) | true.
```

and transform it into kernel GHC in the fashion established for Parlog [5] p.64-69. The transformation of GHC to kernel GHC in order to make all input

matching on head arguments explicit by being performed by extra literals in the guard is a likely part of a compilation strategy for GHC, so it is worth showing how a localised suspension test fits in with such a compilation strategy.

```
(2) head(A) :- [B] <= A, test(B) | true.
```

Here $[B] <= A$ signifies unidirectional unification used for input argument matching. The left hand side is to be unified with the right, but only so that no variable on the right hand side is bound to a nonvariable. In such a case the unidirectional unification suspends until its arguments cannot be unified whereupon it fails, or until it can unify them other than by binding right hand side variables to non-variables.

Replacing B by a fresh variable $B1$ in the user-defined guard *test/1* in (2) and adding a special system predicate *ward/2* to link the two variables in a way that protects the calling environment from being instantiated by suspending the call instead, gets the result:

```
(3) head(A) :- [B] <= A, ward(B1,B), test(B1) | true.
```

On general grounds it can be seen that *ward/2* needs to be able to suspend on several variables because it must handle complex terms. Because complex terms can get progressively instantiated, it also needs to be able to suspend, awake and pass on values and then maybe suspend again. In other words it needs to be able to function transiently. Other properties of *ward/2* can be inferred from the fact that the guard literal *test/1* might need its argument to be instantiated to a value in order for it to succeed. The satisfaction of this guard can only be achieved if the *ward/2* literal will pass bindings to its first argument from its second. However *ward/2* literal should not allow a user defined guard literal to export a value from its satisfaction by having *ward/2* simply unify $B1$ with B . What is wanted is something more like unidirectional unification.

If *ward/2* is equated with $<=/2$'s unidirectional unification, then the second clause of the guard will succeed in cases where it should not. In (3) the calling argument bound with A might get instantiated to $[_I21]$ and the first input matching unidirectional unification succeed sharing B with $_I21$. Since $B1$ and B are still uninstantiated, the second guard safety unidirectional unification can succeed by sharing them. Shortly afterwards the parallel *test/1* call might bind the variable $B1$ to *incorrect*. The result could be the instantiation of a calling argument variable $_I21$ to *incorrect* before the clause has committed. This behaviour cannot be avoided by insisting that the unidirectional unification test

be performed only after the *test/1* literal has succeeded, because then bindings could never be passed through *ward/2* into the user defined guard literal.

A better idea is to make the *ward/2* predicate reluctant unidirectional unification so that the *ward/2* predicate suspends if, in order to unify unidirectionally its arguments, it has to share variables. Reluctant unidirectional unification is unidirectional unification where the process of passing of bindings to the instantiable side carries on while and so long as non-variable bindings arrive on the non-instantiable side and unidirectional unification is possible. It fails if the two sides can never be unified. Where instantiable side variables need to be associated with complex non-instantiable side terms containing variables, they are bound to a consistent copy of 'the complex term to avoid variable sharing. The unidirectional unification is reluctant in that variables on the instantiable side only get instantiated to non-variables. They never get shared with variables on the non-instantiable side. The predicate suspends instead.

The problem now is that the test is too severe. If the guard literal needs a partially instantiated or uninstantiated argument in order to succeed, the *ward/2* predicate must not suspend for ever. For example if the guard literal was defined as follows:

```
(4) test(A) :- true | true.
```

then the result of reluctant unidirectional unification would be for the *ward/2* predicate to suspend with two variable arguments even when the user-defined guard *test/1* has succeeded. What is wanted is for the *ward/2* predicate to suspend on two uninstantiated arguments but only for as long as the guard literal it is warding has not succeeded.

To implement the *wait until the guard literal has succeeded* mechanism, the binding of a variable can be used to signal that the user-defined guard literal has succeeded. The signal can be sent to the reluctant unidirectional unification predicate and the predicate can be defined in such a way that when it receives this signal, it no longer reluctantly unidirectionally unifies its arguments. Instead it tries to restore the linkage between the original variable in the guard literal and its replacement variable by non-reluctant unidirectional unification. This unidirectional unification will ensure that bindings generated by evaluating the guard literal are not passed out to the calling environment and will cause the primitive to suspend should this be attempted. This unidirectional unification is stricter than the $\leq/2$ used for head argument

matching, because of the need to ensure that calling argument variables on the right hand side do not become shared in the unidirectional unification process. In the mapping of Parlog to kernel Parlog input matching on repeated head argument variables is performed separately from `<=/2`, so no constraint of non-sharing of right hand side variables applies to `<=/2`.

Putting these considerations together gives the following. The clause in (1) gets transformed into:

```
(5) head(B) :- [A] <= B,
               ward(A1,A,Control),
               satisfy(test(A1),Control) |
               true.
```

The first literal of the guard is merely to implement head argument matching. The next two literals are there to implement the original guard literal set about by the GHC suspension test. To do this another system predicate *satisfy/2* is introduced. It is a meta-call that attempts to satisfy its first argument. If it succeeds, it binds its second argument to some simple term and succeeds itself. If the attempted satisfaction fails, it fails. It is different from the Parlog meta-call *call/2* which never fails [10], although the meta-call *satisfy/2* can be defined using *call/2*.

The idea of the suspension mechanism in (5) is that up until the user-defined guard literal succeeds, the calling argument variable and its replacement are linked by a reluctant unidirectional unification that does not allow sharing of variables. If and when the guard literal succeeds, then the variable *Control* is instantiated by the metacall *satisfy/2*. This signals to *ward/3* not to suspend reluctantly the unidirectional unification. The relinkage with the original variable is established through this subsequent non-reluctant unidirectional unification that does allow sharing of variables between its two arguments.

4. General Method

Transferring this translation scheme into a general scheme requires taking into account multiple occurrences of unsafe variables in one or more guard literals. The simplest sound way to do this is to place the entire original guard in a *satisfy/2* meta-call if any part of the guard contains an original head argument variable and to use an aggregate term of all unsafe variables and a consistent copy formed by substituting fresh variables for each unsafe variable as the two terms that are to be related by *ward/3*.

The general rule for implementing guard safety suspension in GHC applies to guard literals containing user-defined predicates and system predicates with argument positions liable to bind given values. After it has been applied, head argument matches can be unbuilt into the guard in the fashion for transforming Parlog to kernel Parlog.

- 1) The set of all variables occurring in a head argument and also occurring in the guard in a user-defined literal or in an argument position of a system literal that is liable to be instantiated is formed and called the at risk set. For each member of the at risk set a distinct fresh substitute variable is created. If there are no elements in the at risk set, nothing further is done.
- 2) Every occurrence of a member of the at risk set in a guard literal is replaced by its substitute variable.
- 3) The whole GHC guard is put into the first argument of a *satisfy/2* meta-call and a fresh variable to be called the control variable is put into its second argument.
- 4) A structure with an arbitrary functor and arguments consisting of all members of the at risk set of variables is formed. A copy of this structure is made but with substitute variables replacing at risk variables. A *ward/3* predicate is formed with the formed structure copy as its the first argument, with the formed structure as its the second argument and with the control variable as its third argument. This *ward/3* predicate is added to the guard just in front of the meta-call as another literal in parallel with the others.

5. An Example Translation

An example shows the result of applying the complete transformation from GHC. The following GHC clause

```
(6) process([A|B],C,[A|D]) :- check([A|B],E),
                             test([A|D],E) |
                             process(B,C,D).
```

is translated to the kernel GHC clause

```
(7) process(F,C,G) :- [A|B] <= F,
                      [H|D] <= G,
                      A == H,
                      ward((I,J,K),(A,B,D)),L),
                      satisfy((check([I|J],E),test([I|K],E)),L) |
                      process(B,C,D).
```

The number of literals in the guard has increased from two to five. The first three new guard literals implement input argument matching. The fourth literal implements the safe guard suspension test for all three relevant variables and the fifth literal meta-calls the original guard. Only one extra system literal and a meta-call are required to implement the localised run-time suspension test.

6. A Localised Suspension Algorithm

The algorithm given for implementing *ward/3* is purely sequential. More efficient sequential algorithms exist and parallel algorithms probably exist as well. However it is reasonably succinct and so conveys the basic idea simply. The algorithm returns one of three values each time it is executed success, failure or suspend. It is executed once each time the *ward/3* process is run until it returns *success* or *failure*. When it returns *suspend*, the process evaluating *ward/3* should be hung on appropriate suspension lists for each relevant variable. Details concerning appropriate variables to suspend the *ward/3* process on have been omitted from the algorithm's description but are not hard to determine.

Algorithm

```

clear suspension flag and term catalogue
dereference third argument of ward/3
apply Procedure to the first two arguments of ward/3
IF Procedure returns failure
THEN return failure
ELSE IF third argument of ward/3 is not ground
    THEN IF suspension flag is set OR catalogue is not empty
        THEN return suspend
        ELSE return success
    ELSE IF suspension flag is set OR catalogue term is bound
        THEN return suspend
        ELSE share variable pairs in catalogue and return success

```

Procedure

```

dereference left hand argument L and right hand argument R
IF L is a variable
THEN IF R is a variable
    THEN IF <L, R> is not a recorded pair
        THEN IF <L1,R> is a recorded pair
            THEN bind L to L1
            IF <L,R1> is a recorded pair
                THEN set suspension flag
                IF no <L,R1> or <L1,R> are recorded pairs
                    THEN record pair <L,R> in catalogue
        ELSE IF R is a ground term
            THEN bind L to R
            ELSE copy R to new L1 in light of catalogued pairs adding
                new variable pairs to catalogue and bind L to L1
    ELSE EITHER R is a variable
        set suspension flag
    OR R is an atomic term
        IF L is not same term
            THEN return failure
            ELSE continue
    OR R is a complex term
        IF L does not have same functor and arity
            THEN return failure
        ELSE FOR each argument Li of L
            apply Procedure to Li and Ri
            IF Procedure returns failure
                THEN return failure
            ELSE continue

```

As lists can be represented by two argument structures, they are not handled separately. Where copying is involved, uncatalogued right hand side variables get copied to fresh left hand side variables and the association gets catalogued. The whole Algorithm is executed atomically.

7. Adequacy of Translation Method

No variable that occurs in the head of a GHC clause or in input matches of the $==/2$ or $<=/2$ sort occurs in the guard literals put into the first argument of *satisfy/2*. Thus neither unifying head argument variables with the goal nor performing input matches will bind or share variables in the meta-call's first argument goals. Since no other literal occurs in the clause's guard apart from *ward/3*, only executing *ward/3* can pass bindings into or out of the metacall's goals. Since the metacalled goals are a consistent copy of the original GHC clause guard, their satisfaction conditions for the same input bindings must be the same. In particular, if the satisfaction of any original guard literal fails, then the meta-call *satisfy/2* will fail. What remains to be argued is that *ward/3* consistently passes input bindings yet protects calling argument variables from being instantiated or from being shared with each other.

Before the satisfaction of the original guard literals by the meta-call is complete, whenever a right hand side term, i.e. second argument term in *ward/3*, is discovered bound and its corresponding left hand side term, i.e. first argument term in *ward/3*, is discovered unbound, the unbound term is bound to the bound term or to a copy of it. Thus input values are always passed. The requirement for copying terms consistently ensures this is done consistently. The Procedure does not allow values to be passed from the left hand side to the right hand side and sets the suspension flag if any part of the left hand side is more instantiated than the right hand side. In this way the *ward/3* predicate suspends if guard literals attempt to export values to the call's environment.

Unidirectional unification in *ward/3* can be transient because the only variables that can be instantiated to new values are local to the guard. If the *ward/3* predicate fails after previously making some bindings, no harm is done because the bindings are only local to the guard being warded. However transient unidirectional unification can allow the unwanted communication of bindings if variables are first shared and then a left hand side variable gets instantiated further. What makes this unproblematic here is that no variable sharing between the right hand side and the left hand side is allowed by the Procedure. The only place that allows sharing of right and left hand side variables is in the main part of the Algorithm after the meta-called guard literals have been satisfied and signaled that this is so by instantiating the third argument of *ward/3*. Hence the value in the first argument of *ward/3* is as fully instantiated as it will ever get from satisfying the guard literal. So there can be no sharing of variables between the first two arguments of *ward/3* which the first argument then instantiates further and so communicates unwanted bindings into the

second argument.

8. A Flatter Translation Alternative

This translation of GHC could be made more efficient by analysing the predicate invocation structure of the GHC source program. By analysing whether the occurrence of a head argument variable in a guard literal is being used to supply a value or could export a value, many GHC clauses could be given a simpler translation. The translation method need not always have to provide for mechanisms to enable input uses of the variable in a guard literal and yet protect the calling environment against instantiation at the same time with a suspension test should satisfaction of the guard literal result in the attempt to bind that variable with a value.

If the guard argument in (1) was analysed to be an input argument that would not bind any part of the argument, then (1) could be translated to the following:

```
(8) head(B) :- [A] <= B, test(A) | true.
```

In this case there would be no need for the suspension test. The clause (1) and the clauses for *test/I* would be part of safe GHC.

However such a translation method could not sustain the claim that the method of translation preserved the execution conditions for each GHC clause. This method of translation would only ensure that the execution of a set of translated clauses had the same result as executing the original clauses. If the clauses for any user-defined guard literal were changed so that its mode of use of a head argument changed, the translation would no longer be valid. For example the clauses for *test/I* might be changed so that its original form

```
(9) test(correct) :- true | true.
```

was changed from being a user-defined guard literal that expects an input argument to a literal that tries to bind a calling argument.

```
(10) test(A) :- true | A = correct.
```

As Miyazaki has been reported to observe in a private communication [4], such a method of translation would not allow the modular development of GHC programs. However, the translation method described here preserves the execution conditions of each GHC clause. Even if the definition for a user-defined guard predicate changes the mode of use of argument variables, a new

translation of the original GHC clause will not have to be provided. Thus modular program development is not affected by the proposed translation method.

9. Implementing Localised Suspension

Implementing *ward/3* requires allowing a single process evaluating it to be simultaneously suspended upon several variables. If an implementation only allows a process to be suspended upon one variable at a time, then the same effect can be had by spawning a suitable child process for each relevant variable, suspending each child process upon its variable and suspending the originating process upon its children. By arranging for an execution scheme whereby processes suspended upon a variable are scheduled and run immediately that variable is shared or instantiated, the latency of the suspension test mechanism can be minimised.

Because the meta-call *satisfy/2* does not need to wait until its first argument is instantiated, because this argument is only used when its argument is instantiated to a literal or a conjunction of literals, the goals in its goal call can be scheduled at the same time as the meta-call. In fact they are best scheduled before their parent call *satisfy/2*, so that in a typical bounded depth first scheduling implementation, not very deep, guards will tend to be satisfied or fail before their parent is ever run thus minimising the meta-call's latency.

On a compiled implementation with suitable instructions it is possible to dispense entirely with the need for a meta-call. Instead of an EXECUTE instruction for *satisfy/2* the compiled code for the call *satisfy(test(A),B)* could be

```
CALL    test/1
ASSIGN a2 0
```

Previous instructions are assumed to have prepared and loaded appropriate argument registers. The CALL instruction calls the code for *test/1* and returns execution to the following statement upon successful completion, otherwise it returns control on failure depending upon where it was called from. The instruction ASSIGN to argument register a2 of the integer value zero instantiates it to a ground term. In this way the required sequencing of the signaling action can be achieved through sequential execution of the compiled code instructions without creating an extra meta-call process. Handling the GHC suspension test by localisation is part of an implementation scheme for

GHC and Parlog for a single processor implementation. A two way translation interface maps these languages into an and-parallel subset of kernel Parlog and back. The results of translations are emulated directly [11].

10. Conclusion

A technique has been described for translating individual GHC clauses to individual clauses in a safe variant language, the parallel subset of kernel Parlog, while preserving their execution conditions. The existence of the technique helps to illuminate the relationship between GHC and Parlog. Furthermore by localising GHC suspension to the evaluation of the responsible guard, the overhead of implementing it is imposed only upon guard evaluations involving unsafe variables. The technique allows the execution of full GHC on Parlog implementations with only moderate extensions. The technique can also be used straightforwardly on Parlog implementations to help with ensuring guard safety where a Parlog compile time safety check is undecidable. The technique allows more effective use of computation space in full GHC implementations by preventing suspension effects from propagating outward into output bindings performed during evaluation of deep guards resulting in the suspension of whole parts of the tree of evaluation of that guard.

Acknowledgements

The author would like to acknowledge financial support from Alvey project IKBS 90, help from Imperial College's Parlog group, and the support of his wife and colleagues at Heriot-Watt University.

References

1. Kazunori Ueda, "Guarded Horn Clauses," Doctor of Engineering Thesis, University of Tokyo, Graduate School, Tokyo, (March 1986).
2. V. Saraswat, "GHC: Operational semantics, problems and relationship with CP(\downarrow , $|$)," Proceedings of the 1987 Symposium on Logic Programming, Computer Science Department, Carnegie-Mellon University, San Francisco, USA, (September 1987).
3. Jacob Levy, "A GHC Abstract Machine and Instruction Set," Proceedings of the Third International Logic Programming Conference, Springer-Verlag, London, (July 1986).

4. Masasuke Kishi, Eiji Kuno, Kazuaki Rokusawa, and Noriyoshi Ito, "The Dataflow-based Parallel Inference Machine To Support Two Basic Languages in KL1," ICOT Technical Report 114, (July 1985).
5. Steve Gregory, Parallel Logic Programming in Parlog - The language and its Implementation, International Series in Logic Programming, Addison-Wesley Publishing Company, London, (February 1987).
6. E. Shapiro, "A Subset of Concurrent Prolog and its Interpreter," ICOT Technical Report TR-003, Institute for New Generation Computer Technology, Tokyo, Japan, (February 1983).
7. E. Shapiro, Concurrent Prolog: A Progress Report, Technical report CS86-10, Department of Computer Science, The Weizmann Institute of Science, Rehovot, Israel, (April 1986).
8. Akikazu. Takeuchi and Koichi Furukawa, "Parallel logic programming languages," Proceedings of the Third International Logic Programming Conference, Springer-Verlag, London, (July 1986).
9. J. Levy and E. Shapiro, Translation of Safe GHC and Safe Concurrent Prolog to FCP, Technical report CS87-08, Department of Computer Science, Weizmann Institute of Science, Rehovot 76100, Israel, (June 1987).
10. K. Clark and S. Gregory, Notes on Systems Programming in Parlog, Research Report DOC 84/15, Department of Computer Science, Imperial College, London, (April 1984).
11. H. Taylor, "Kelpie - An And-Parallel Kernel Parlog Emulator," Technical Report 87/11, Computer Science Department, Heriot-Watt University, Edinburgh, (December 1987).