

Coupling Committed and Trial Binding Resolution Engines

Hamish Taylor

Queen Mary and Westfield College

University of London

The ability of sequential and parallel Prologs to implement don't know non-determinism in resolution efficiently makes them apt for supporting a knowledge base querying capability. On the other hand their search based execution model and their trial use of bindings makes them unsuitable for systems programming. Conversely the use by concurrent logic programming or CLP languages of committed bindings, general and-parallelism and synchronisation constraints enables them to support systems programming applications quite well. However, their use of committed choice resolution makes them unable to support a knowledge base querying capability. These limitations of the main logic programming schemes make it awkward to program concurrent knowledge based systems in logic. That would require the means for sustaining both knowledge base querying and systems management capabilities within a single logic programming scheme. However, the querying parts of a concurrent knowledge based system could be realised with trial binding computations on a multi-threaded Prolog engine and the systems programming parts with committed binding computations on a CLP engine by loosely coupling these engines together. This paper proposes suitable interfaces and methods for realising these forms of coupling. A coupled meta-interpreter and multiple solutions predicates illustrate CLP-Prolog programming on a coupled system.

Keywords: concurrency, knowledge base, systems programming, search

1. Introduction

If concurrent knowledge based systems are to be supported within a single logic programming implementation then that implementation must be able to sustain two different kinds of capability

- knowledge base querying
- systems programming

An adequate knowledge base querying capability requires the the ability to perform complete and terminating searches of its stored knowledge. Under certain assumptions sequential and parallel Prologs can do this efficiently, because they can exhibit don't know non-determinism in resolution [10, 18]. However, the concurrent logic programming or CLP languages do not exhibit don't know non-determinism, and so cannot support an adequate knowledge base querying capability directly. Furthermore, they cannot efficiently support general don't know non-determinism indirectly, even

by compilation using continuations in GHC [17], meta-interpretation using copying in Parlog [4], meta-interpretation in FCP [14], or partially evaluated meta-interpretation using substitution libraries in FGHC [7].

Some forms of interacting systems can be modelled by resolution strategies which allow a fine grained process interpretation to be given to logic programs, so long as individual goals can be delayed on data-flow synchronisation constraints and some form of co-routining of conjoined goals is supported. CLP languages are the most expressive way of doing this, but analogous capabilities can be demonstrated by and-parallel or co-routining Prologs supporting goal delay mechanisms like Prolog-II's freeze mechanism [3]. However, a suitable interpretation of logic program execution for general systems programming requires variable bindings to be deterministic vehicles for sending messages. This conflicts with using them as trial values in a don't know non-deterministic search. Thus the capacity to exhibit don't know non-determinism conflicts with the kind of interpretation which needs to be given to parts of executing logic programs to view them as concurrent communicating system components and in practice seriously undermines modular, scrutable systems programming. The efficient implementation of search by backtracking or or-parallelism also interferes with dependent and-parallel execution of goals (i.e. goals with shared variables) because of the difficulty of efficiently coping in a search based execution model with conflicting bindings being made to shared variables by goals executed in and-parallel [6].

Furthermore, co-routining and and-parallel Prologs, like NU-Prolog [12] are no more expressive than the flat CLP languages like FCP [13] in exhibiting stream and-parallelism and are incapable of sustaining other important kinds of systems handling capability required by concurrent knowledge based systems like fair scheduling of sub-tasks within the whole computation. The same applies to other attempts to reconcile stream and-parallelism with don't know non-determinism like the lazy non-determinism of Andorra Prolog [9] and Pandora [2] and P-Prolog's use of the exclusive relation [19]. Thus while Prolog-like languages can program the knowledge base querying aspects of concurrent knowledge based systems by exploiting don't know non-determinism, and while the CLP languages can sustain systems programming capabilities using stream and-parallelism, none of these possibilities can sustain the combination adequately in order to program concurrent knowledge based systems in logic. This suggests forming a hybrid of both resolution strategies by coupling CLP computations with Prolog computations to achieve each of the required capabilities together. The approach will preserve efficiency if it can do this without interfering with the basic mechanics of each type of resolution engine.

2. Parlog and Prolog United

Clark and Gregory first advocated coupling resolution engines exhibiting don't know non-determinism and stream and-parallelism in terms of interfacing Parlog with Prolog [5]. They defined six interfaces. Three interfaces enable a Parlog computation to call a Prolog computation

- eager all solutions predicate `set(List^, Term?, Conj?)`
- lazy multiple solutions predicate `subset(List?, Term?, Conj?)`
- single solutions predicate `prolog_call(Conj?)`

`set/3` and `subset/3` are eager and lazy multiple solutions constructors like Prolog's `findall/3`. They deliver solutions instances of *Term* to the goal *Conj* incrementally as elements of *List*. `subset/3` binds solution elements of *List* as variable list elements are given to it whereas `set/3` extends *List* autonomously. The third interface `prolog_call/1` can have its variables bound at any time during its execution either by other Parlog goals executing concurrently with it or by the Prolog computation it represents. These new bindings may not be rescinded by the Prolog computation once made. Since the multiple solutions constructors can be defined using `prolog_call/1` (and destructive assignment), it embodies the functionality underlying all three interfaces. Three interfaces are also defined which enable Prolog to call Parlog

- deterministic conjunction `prolog-conj :: parlog-conj`
- eager non-deterministic conjunction `prolog-conj <> parlog-conj`
- lazy non-deterministic conjunction `prolog-conj << parlog-conj`

Each represents co-routining conjunctions. `::/2` spawns the Parlog conjunction immediately on execution and continues executing the Prolog conjunction. The Prolog conjunction may engage in backtracking so long as no bindings passed to the Parlog conjunction are undone. `::/2` succeeds when both conjuncts succeed. When the Prolog conjunct is *true*, Prolog is just synchronously invoking Parlog. The second and third interfaces allow failures in the Prolog conjunction to fail and undo bindings to variables shared with the Parlog conjunction. This rolls back the Parlog computation to the point at which the uncommitted binding, which was undone, was made. If the Parlog computation itself fails, the goal which caused the most recent uncommitted binding in the Prolog conjunction is supposed to be failed. `<>/2` allows the Prolog conjunction to carry on making uncommitted bindings to variables shared with the Parlog computation eagerly. `<</2` allows the Prolog conjunction to make deterministic bindings to shared variables eagerly yet delays making a binding it could undo on backtracking, and only proceeds if and when the Parlog conjunction deadlocks.

Clark and Gregory want to explore ways of coupling Parlog and Prolog together with a view to stimulating further research into the design, implementation and use of hybrid don't know and don't care non-deterministic logic programming systems. The non-deterministic interfaces `<>/2` and `<</2` represent the really radical departure in their paper. They entail extending Parlog to allow bindings in a Parlog computation to be undone and the computation rolled back. Clark and Gregory devote most of their attention to these non-deterministic operators. Of the two only `<</2` affords sufficient scope for control to be of interest. However, the motivation for the research described here is not to explore the full range of possible interfaces between Prolog and CLP computations, but to support concurrent

knowledge based systems in logic. This kind of application mostly concerns calling Prolog-like computations from Parlog-like computations. However, interfaces which enable Prolog computations to invoke CLP ones are also useful to allow knowledge bases realised in Prolog to deal with concurrent interfaces.

Clark and Gregory advocate a shared memory approach to storing variables shared between Prolog and Parlog computations, [5] section 4. They allow each of a coupled pair of Parlog and Prolog computations to bind variables in each others heaps. A non-shared memory approach would make each Prolog and CLP computation maintain a separate binding environment and copy bindings across the interfaces both ways. A shared memory approach is less desirable for several reasons.

- efficient variable management is interfered with
- garbage collection becomes rather problematic
- loosely coupled multi-processing is made inefficient
- modular development of coupled systems is undermined

Avoiding creating dangling pointers in resolution implementations depends on determining the direction of variable to variable bindings by their address magnitudes. If bindings are allowed across multiple extra heaps, simple magnitude tests cannot enforce variable binding direction policies. For the same reason garbage collection is made much harder, because an intertwined tangle of variable pointers across multiple heaps must be traversed before dead space can be safely identified. Loosely coupled multi-processing is also desirable for concurrent execution of coupled CLP and Prolog computations, especially to achieve highly parallel execution. Reducing avoidable sharing of memory areas is crucial to enhancing its efficiency. Lastly, a coupling strategy should be modular to make it easy to keep abreast of developments in resolution engine implementation technology. Thus it should only require minor adaptations to the workings of each engine of a coupled system. Otherwise a coupled system is liable to become a specialised idiosyncrasy which is bypassed by developments. For these reasons it would be better to make all bindings local to their own computation, and to copy changes to shared bindings across between computations.

3. Coupled Resolution Engines

Concurrent knowledge based systems can be realised by invoking multiple Prolog computations from a CLP computation. Each Prolog computation can handle a separate knowledge processing task, and the CLP computation can manage task interaction and handle any other task requiring significant communication, control or interaction.

Three issues affecting the design of suitable computation interfaces are:

- concurrency of components
- revocability of shared bindings

- view of database clauses

3.1. Concurrency

Concurrency between multiple computations can be sustained in different ways. The coupling interfaces require that multiple CLP computations be able to execute concurrently with multiple Prolog computations. Multiple CLP computations can already co-exist unproblematically on CLP implementations which support a reasonably versatile meta-call. Each separate computation can be executed using a different meta-call [8]. This allows a fairly fine grain of concurrency.

Different means for supporting concurrency among many Prolog computations are needed, because canonical Prolog implementations do not support concurrent execution of goals, and co-routining or independent and-parallel Prologs cannot support fair concurrent execution of goals. Multi-threaded Prolog execution can be supported using Unix's concurrency by running several different processes executing Prolog at the same time using pipes, files or shared memory to communicate among them. Parallel Unix implementations like Dynix [1] make this strategy viable for shared memory multi-processors as well. However, such an approach is not efficient for managing a system of closely interacting and communicating tasks. It embodies too coarse a grain of concurrency. It is inflexible to changes in processing grain because of the start up and shut down overheads of creating or terminating a new Unix process for executing Prolog. It also imposes restrictive bandwidth limits on inter-Prolog communication.

However, having access to a CLP resolution engine means that those tasks of a concurrent knowledge based system which may execute at the same time with a widely varying grain of concurrency but do not require don't know non-deterministic execution, can be executed on the CLP resolution engine. This includes interface management, input/output handling, systems coordination, task scheduling, and other control related jobs. Only tasks like query satisfaction, integrity checking, explanation construction and perhaps query optimisation require don't know non-deterministic execution. However, in typical applications they will not be required to interact tightly with other tasks. Thus coarse grained concurrency in executing sequential Prologs would suffice to process them.

This suggests supporting a concurrent knowledge based system on a coupled CLP-Prolog system by using one Unix process (or several tightly interacting Unix processes sharing memory on a multi-processor) to execute the CLP language and several Unix processes to execute sequential Prologs. Unix sustains overall concurrency among the multiple sequential Prologs and the sole CLP engine. The CLP resolution engine can process the closely interacting and communicating tasks of the knowledge based system as CLP computations with fine granularity concurrency. The remaining knowledge processing tasks requiring don't know non-determinism can be processed with low degrees of interaction and coarse granularity concurrency by separate sequential Prologs. The whole system executes a CLP

program on the CLP engine and separate Prolog programs in each Prolog process.

However, it would also be desirable to sustain concurrent execution of multiple Prologs on a more closely coupled basis. This could be realised under System V Unix using its shared memory facility or under Sun OS 4.0 [11] using lightweight processes sharing the memory of a single Unix process. The code area and symbol tables could be shared between lightweight processes using a concurrency control mechanism such as monitors to handle updates to these areas, and the rest of shared memory could be divided up to give each separate Prolog computation thread its own local, global and trail stacks.

3.2. Revocability of Shared Bindings

A simple way of reconciling the differences between a CLP and a Prolog implementation, without undermining either Prolog's efficient stack based implementation, or a CLP computation's lack of need to restore prior states of the computation is to insist that Prolog bindings made to variables shared with a CLP computation cannot be revoked once they are communicated.

Irrevocable Communication bindings communicated to a CLP computation cannot be undone
by the Prolog computation

Violating this restriction fails the Prolog computation. This requirement lets a Prolog binding to a shared variable be revoked so long as it has not been communicated. This restriction fits in better with a loose approach to sharing variables using two way copying across the CLP-Prolog divide rather than with Clark and Gregory's tight approach to sharing variables via common memory. The restriction allows the *Irrevocable Communication* restriction to be enforced by the interface independently of the internal workings of the CLP and Prolog engines, which is important for modularity.

Binding irrevocability across interfaces would rule out complex interfaces like Clark and Gregory's non-deterministic invocations of Parlog from Prolog $\langle \rangle / 2$ and $\langle \langle / 2$. Three reasons for doing this are to

- keep semantic simplicity in the interface constructs
- preserve the systems programming model of CLP languages
- retain efficiency in the implementations coupled together

Allowing revocable bindings to be passed across the interfaces from Prolog to CLP computations would make the interface semantics rather complex. Revocable bindings interfere with the role given to variables in the systems programming interpretation of logic program execution of being the message passing medium for concurrent, communicating processes. Messages, once sent, have happened and cannot be undone. However, if revocable Prolog

bindings can be shared with a CLP computation and then rescinded in both, the CLP computation could be subject to such non-determinism. CLP-Prolog applications would be subject to a strange kind of retroactive interference which would rescind messages sent between sub-systems, as if they had never happened. Thirdly, by not allowing Prolog computations to rescind communicated bindings and rollback CLP computations which have consumed and acted upon that binding, the efficiency of existing approaches to implementing CLP resolution engines can be preserved.

3.3. View of Database

For adequate flexibility a coupling scheme of CLP computations to Prolog computations should allow Prolog computations to have

- private and shared databases
- persistent inheritable or uninitialised new database states

In order to allow for the simple manipulation of these recommended relationships of Prolog computations to clause databases, the following principle will be adopted.

One Database per Meta-call each separate CLP meta-call environment is associated with a separate persistent Prolog database state.

This database state is created by the first call to Prolog from within a meta-call environment. After that, each call to Prolog within that meta-call environment accesses the same database state for as long as the meta-call environment persists. Prolog invocations within different CLP meta-calls always access different database states. This rule is simple to grasp and use. It can be realised using lightweight processes with a common code area and symbol table in a real Unix process to enable concurrent sequential Prologs to share a database, and using distinct real Unix processes to create private database states.

4. Computation Interfaces

In what follows an integrated set of ways are proposed for coupling CLP and Prolog systems. They presuppose that the environments of the two resolution engines are separate. All communication across interfaces is irrevocable and involves copying bindings across. In this way the versions of the goals on each side remain able to unify, but do not store shared variables in a commonly accessed memory. Each communication is atomic, with no bindings being made if the goal versions cannot unify. If such a unification is impossible, the invoked computation fails with normal goal failure consequences for the invoking environment. Two interfaces allow Prolog to call a CLP computation as follows:

call to CLP relation

- communication at boundaries
- input argument suspension on invocation possible
- synchronous

call to *clp(Goal)*

- communication transiently both ways (with a restriction)
- input argument suspension on invocation of Goal possible
- asynchronous

Calls using these interfaces are never resatisfiable. Goals in a Prolog computation, which are not defined by a clause relation and are not calls on a primitive, invoke the CLP resolution engine. The invocation is synchronous, and values are passed at the boundaries of invocation without any concurrency between the calling computation and the computation which is called. Input matching suspension on invocation takes place if needed. This interface is essentially a version of the interface proposed by Clark and Gregory's deterministic meta-conjunction operator *true :: relation(Arg1,..., Argn)* where *relation/n* is a CLP relation. Calls in Prolog to *clp/I* will invoke the CLP resolution engine asynchronously. Argument values may be passed transiently during invocation subject to the following restriction on shared variables

No Complex Term CLP Bindings CLP execution may not bind variables shared with a Prolog computation to complex terms during execution

A CLP computation is only allowed to bind shared variables to simple terms when executing a *clp/I* call. The interface is responsible for detecting the attempt to communicate to a Prolog computation the binding of a shared variable to a complex term. This restriction is explained later, but is essential for solving a memory management problem with concurrent bindings in a canonical Prolog engine. *clp/I* goals in Prolog coroutine with all siblings goals in its clause body, or if invoked at the top level with all other top level goals. To keep the scope of its concurrency simple, *clp/I* goals are not allowed to appear in goal bodies with a disjunction ; or a conditional operator <-. The parent goal of the spawned call to the CLP resolution engine does not exit until the *clp/I* call has succeeded or failed. When a *clp/I* goal fails, Prolog backtracks to look for another satisfier of the parent goal. The behaviour of *clp/I* is similar to Clark and Gregory's deterministic *::/2* operator, although no scoped conjunction operator has been introduced.

Two asynchronous interfaces allow a CLP computation to call Prolog as follows.

call to Prolog relation

- communication at boundaries

- executes concurrently with CLP resolution engine
- no suspension on invocation

call to *prolog(Goals)*

- communication transiently both ways (with a restriction)
- executes concurrently with CLP resolution engine
- input argument suspension if *Goals* unbound

CLP goals, which are not defined by a guarded clause relation and are not calls on a primitive, invoke the Prolog resolution engine. Invocation values are passed at the boundaries of invocation with full concurrency between the calling and the called computation. No suspension on invocation takes place. Calls in a CLP computation to *prolog/I* invoke the Prolog resolution engine asynchronously in a fully concurrent fashion. Input and output bindings are communicated transiently except that the *No Complex Term CLP Bindings* restriction applies to the CLP computation executing concurrently with *prolog(Goals)*. The CLP computation is only allowed to bind shared variables in *Goals* to simple terms when executing a *prolog/I* call. The interface is responsible for detecting the attempt to communicate to a Prolog computation shared variables bound to complex terms by the CLP computation. The call to *prolog/I* suspends on its argument, if it is unbound. *prolog/I* is rather similar to Clark and Gregory's proposed interface *prolog_call/I*, although the intended implementation approach is quite different.

5. Implementation Issues

Communication between the CLP and the Prolog engines can be achieved at various levels. High level interfaces will be described, because they demonstrate the functionality most readily. They prove that coupling is possible with low degrees of intervention into the mechanics of each engine. A simpler variant of the language Parlog will be used to express the CLP clauses. It differs from Parlog by being purely and-parallel, by eschewing modes and input matching instead on all head arguments, and by sequencing input matching on head arguments before guard goal evaluation. In all other respects it conforms to Parlog's execution model [15]. Invocation relationships will be further clarified by introducing two new primitives.

- *clp_machine*(Name, Goals, Result)
- *prolog_machine*(Name, Goals)

clp_machine/3 is a non-resatisfiable Prolog primitive which presupposes that *Goals* is ground (apart from occurrences of *Name*) when invoked. Satisfying it, invokes the CLP engine, unifies *Name* with a unique identifier of a file-like interface to the CLP engine, and starts executing *Goals* in a separate meta-call environment concurrently with any existing CLP engine goals. It associates *Result* internally with the CLP computation so that it is unified with *yes* if

and when the CLP computation ends successfully. *prolog_machine(Name, Goals)* is a CLP primitive which suspends until *Goals* is ground (apart from occurrences of *Name*). It then creates a new Prolog computation thread, unifies *Name* with a unique identifier of a file-like interface to it, and executes *Goals* in that thread. These primitives elicit the names of file-like interfaces for communicating between CLP and Prolog computations. They can be realised by pipes, sockets or shared memory buffers. Buffered communication will be achieved between CLP and Prolog computations by reading and writing to these file-like objects. A Prolog computation will be assumed to wait doing nothing until input arrives to satisfy a *read(Term, File)* primitive call, whereas a CLP computation will be assumed to suspend the primitive goal *read(Term, File)* and carry on executing other CLP goals, or sleep if deadlocked, until input on *File* wakes up the suspended goal *read/2*.

In terms of Sun's OS 4.0 Unix [11] multiple Prolog and CLP computations interact as follows:

- CLP calls in Prolog invoke different CLP meta-calls in one CLP engine
- Prolog calls in different CLP meta-calls invoke different Unix processes
- Prolog calls in one CLP meta-call invoke different lightweight processes in a Unix process

CLP meta-calls mean Parlog-like stream controlled meta-calls [8] and not simple meta-calls like *call/1*. The CLP engine will be described in what follows as a single Unix process. However, it could also be realised as multiple tightly coupled Unix processes sharing memory. A Prolog computation will either be a single threaded ordinary Unix process with its own local memory, or be a lightweight Unix process within an ordinary Unix process which shares the memory of the Unix process with other lightweight processes (computation threads). Single threaded Unix processes executing Prolog will be standard WAM engines, capable of being forked into several lightweight processes which each maintain their own trail, stack and heap, and use a common symbol table and code area. Separate threads synchronise their updates of the common memory like the code area and symbol table by using monitors on critical code.

5.1. Prolog calling CLP Atomically

A simple synchronous invocation from a Prolog computation to a CLP computation by the goal *relation(Arg1,..., Argn)* can be achieved by replacing it with the goal *guarded(relation(Arg1,..., Argn))*, where *guarded/1* is defined by the following Prolog clause

```
guarded(Goal) :- clp_machine(Name, solve(Name), _),
                write(Goal, Name),
                read(Goal, Name).
```

The primitive *clp_machine/3* has been defined above. *read(Term,File)* reads a *Term* from *File*. *write(Term,File)* writes a *Term* to *File*. *guarded/1* uses the following CLP clauses:

```
solve(Name) :- read(Goals, Name), call(Goals) | write(Goals, Name);  
solve(Name) :- write('$fail', Name).
```

read(Goals, File) unifies *Goals* with the next term read from *File*. *call(Goals)* suspends until its argument is bound before attempting to satisfy *Goals*. *write(Term, File)* suspends until its arguments are bound before writing *Term* to *File*. The CLP goal *solve(Name)* with *Name* bound will be suspended on the two guard goals for the first clause for *solve/1* until satisfaction of the goal *guarded/1* succeeds in sending the term *relation(Arg1,..., ArgN)* to the CLP system. This will be received by the CLP engine and executed as a goal on the CLP engine by the primitive *call/1*. If *call(Goals)* fails, a term signaling failure is written back to the Prolog system, otherwise the satisfied goal is written back. The Prolog system is meanwhile waiting to read the output from the CLP system. When the Prolog system reads the term output as a result of the CLP execution, it attempts to unify the result with the original goal. The original Prolog goal succeeds or fails with that unification.

5.2. Prolog calling CLP Incrementally

The concurrent incrementally communicating interface *clp/1* needs a new Prolog primitive *async(Name,Goal)* for handling communication with the CLP computation. This primitive behaves like the *freeze(X,Goal)* primitive [3] except that *Goal* is woken by input being received by the Prolog computation on the file *Name*. Each time *async/2* is woken up, its *Goal* is executed but it remains suspended, and it is only removed if its point of creation is backtracked across. Furthermore delayed goal scheduling ensures that only one goal delayed by an *async/2* call may be woken and reduced at a time. The primitive *async/2* can be efficiently implemented on WAM engines along the same lines as *freeze/2* [3]. The behaviour of *clp/1* can now be supported at a high level by transforming rule bodies in which *clp/1* occurs. If a *clp/1* goal occurs in a rule body as follows

```
relation(X, Y, Z) :- try(Y, V), clp(test(X, V)), check(Z, V).
```

then it is replaced by a rule which swaps the *clp/1* goal with a *clp_call/2* at the start of the body.

```
relation(X, Y, Z) :- clp_call(test(X,V), Result), try(Y, V), check(Z, V), data(Result).
```

data/1 is a primitive which makes Prolog execution delay until its argument is bound. It ensures that the Prolog computation waits upon the CLP computation finishing. The *No Complex Term CLP Bindings* restriction defined earlier copes with the difficulty of where to put any complex terms which are transiently communicated from the CLP

computation to the Prolog computation during the satisfaction of the concurrently executing Prolog goals - in our example *try/2* and *check/2*. The place to build complex terms in a Prolog engine is on the top of the Prolog engine's heap. However, transiently communicated bindings from the CLP computation, which are stored at the top of the Prolog engine's heap at the time of communication, would be vulnerable to being inappropriately popped after that time by local backtracking by sub-goals of the concurrently executing Prolog goals. There is no other satisfactory place to put such bindings without instituting major changes to the design of the Prolog engine. So it is better to proscribe the making of complex term bindings to shared variables by the CLP computation, and to make the CLP-Prolog interface enforce that restriction. A binding of a variable to a simple term can be done by writing the term on top of the variable. Thus such bindings would be safe to local backtracking, so long as these bindings are not trailed at the top of the trail stack at the time of communication. Instead a trail entry for each shared variable should be created at the time of initial execution of the *clp_call/2* call, so that any communicated bindings are undone should goal failure cause backtracking right back to the parent of the *clp_call/2* goal.

The relation *clp_call/2* is defined in Prolog as follows:

```
clp_call(Goals, Result) :- clp_machine(Name, invoke(Name), Result),
                          otherwise(Name, fail),
                          async(Name, talk(Goals, Name)),
                          write(Goals, Name).
```

On executing *clp_call(Goals, Result)*, the shared variables in its arguments are trailed, and the goal is reduced to its body goals. The first body goal *clp_machine/3* creates an interface to the CLP engine and executes *invoke/1* through it. The implementation also notes the variable *Result* for unifying with *yes* should the CLP computation succeed. An escape exception handler *otherwise/2* for the exception *Name* is then set to catch a failure in the execution of the *Goals* on the CLP engine, which will be handled by raising the exception *Name* [16]. It is instigated by executing the special exception raising primitive *fail/1* which starts deep backtracking at one go across ordinary choice-points to the first special choice-point labelled by the argument of *fail/1* - in this case created by *otherwise(Name, fail)*. The alternative branch (i.e. the second argument of *otherwise/2*) is then executed instead of the rest of the clause body. After the escape exception handler is set up a persistent delayed goal *talk/2* is created using the primitive *async/2* for handling communication with the CLP engine. The goal for CLP execution is then sent to the CLP engine, and Prolog execution continues with the two concurrent Prolog goals *try/2* and *check/2*.

Each interface between CLP and Prolog computations records which computation is responsible for which bindings to shared variables. This library is used to enforce the *Irrevocable Communication* restriction. Transient communication is handled by the persistent suspended goal *talk/2* defined as follows:

```
talk(Name, Goals) :- prolog_bindings(Goals, Name)
                    read_goal(Goals, Name),
                    clp_bindings(Goals, Name),
                    write(Goals, Name), !.
talk(Name, _)      :- fail(Name).
```

Two primitives *prolog_bindings/2* and *clp_bindings/2* record in their private library associated with the interface *Name* any unrecorded bindings to variables in their first argument. Each library entry associates the binding with the Prolog or the CLP side of the *Name* interface responsible for it. If *prolog_bindings/2* notices that recorded Prolog bindings have been rescinded, it fails. Otherwise both predicates succeed. The communication from the CLP engine is read and *Goals* is unified with it using a special primitive *read_goal/2*. *read_goal(Goals, Name)* only succeeds if *Goals* can unify with the term read, and is able to do so without binding variables in *Goals* to complex terms (*No Complex Term CLP Binding* restriction). Furthermore if *read_goal/2* can succeed, it unifies its first argument with the term read, performing any bindings it makes without trailing them on the trail stack by copying simple terms on top of variables. *Goals* is then written to the CLP computation again. Should any of this fail, an escape exception is raised by the second clause for *talk/2*. This results in the handler in *clp_call/2* catching it and then failing execution back to the original goal. In our case the original goal invoked the clause for *relation/3*, and its resatisfaction would result in an alternative satisfier being sought.

The CLP clauses for *invoke/1* assume the existence of a stream controlled meta-call

```
meta(Goals, Resources, Status)
```

for invoking the CLP resolution engine concurrently with other invocations. This CLP primitive executes its *Goals* argument on the CLP engine for one quota of resources for every *quota* element on its *Resources* stream. It either reports using its *Status* argument as a stream each time it has used up a quota by *used_quota*, or reports that it has *succeeded*, *failed*, *woken* or *deadlocked*. In the event of having run out of resource quotas, *meta(G,R,S)* suspends on the variable tail of its resources stream *R*. In the event of a deadlock, *meta(G,R,S)* suspends on all top level variables in *G*. If any of these are instantiated, or if its meta-environment receives a *wake-up* inter-process signal from a Prolog computation to which it is interfaced, *meta/3* wakes up and adds a *woken* token to its status stream. It then waits for the next persistent delayed goal communication to complete on the associated Prolog computation, and adds a new *deadlocked* token to its *Status* stream, if the Prolog computation remains deadlocked.

```
invoke(Name) :- read(G, Name),
               meta(G, [quota|L], S),
               control(G, Name, L, S).
```

```
control(G, Name, L, [deadlocked|S]) :- write(G, Name),
                                       read(G, Name) |
                                       control(G, Name, L, S).
control(G, Name, L, [woken|S])      :- write(G, Name),
                                       read(G, Name) |
                                       control(G, Name, L, S).
control(G, Name, L, [succeeded|S]) :- write(G, Name).
control(G, Name, L, [used_quota|S]) :- write(G, Name),
                                       read(G, Name) |
                                       L = [quota|Ls],
                                       control(G, Name, Ls, S) ;
control(G, Name, L, [_|S])          :- write('$fail', Name) |
                                       fail.
```

Transient communication with the CLP engine is achieved by writing and reading back the current values of goals and unifying the terms read with their own version of the goal. This is done each time the CLP computation deadlocks, or executes a quota of resources, or its control meta-call *meta/3* is woken. The coupled system copes with mutual deadlocks, when a Prolog computation discovers it is deadlocked, by having the Prolog computation pass *wake-up* inter-process signals to processes with which it is interfaced. This continues on a periodic basis until the Prolog computation is itself woken up again. These *wake-up* signals are ignored by CLP computations unless they have deadlocked *meta/3* meta-calls.

5.3. CLP calling Prolog Atomically

A simple call to a Prolog relation in a CLP program, using the CLP-Prolog scheme for combining CLP and Prolog execution, passes values at the start and end of its invocation. Thus the reduction of the goal is an atomic operation from the viewpoint of the CLP computation. If the CLP computation invokes the Prolog relation *relation/n*, then the invoked Prolog goal can be replaced by the CLP goal

```
horn(relation(Arg1,..., Argn)).
```

The CLP relation *horn/1* can be defined as follows

```
horn(Goal) :- prolog_machine(Name, solve(Name)),
              write(Goal, Name) |
              read(Goal, Name).
```

A Prolog computation thread is established, the *Goal* is written to the Prolog system, and the result is read back from the Prolog system and unified with *Goal*. The Prolog clause *solve/1* can be defined as follows:

```
solve(Name) :- read(Goal, Name),
               call(Goal),
               write(Goal, Name).
solve(Name) :- write('$fail', Name).
```

The goal is read, executed, and either written back if successful, or a reserved atom which cannot unify with the goal is written back instead.

5.4. CLP calling Prolog Incrementally

The incrementally communicating interface *prolog/1* is rather more complex to realise. In order to support transient communication the Prolog engine must send and receive at appropriate junctures the current bindings of the common goal. In order to achieve this a stream controlled primitive

```
wam_machine(Name, Call, Resources, Status)
```

is introduced for creating and scheduling execution of a Prolog engine. This CLP primitive suspends until *Call* is ground apart from occurrences of the variable *Name* and until *Resources* is bound. It then creates a new Prolog computation thread, unifies *Name* with the unique identifier of a file-like interface to it, and executes the goal *Call* in that computation thread for one quota of reduction steps for every quota element on its *Resources* stream. It either reports using its *Status* argument as a stream each time it has used up a quota by *used_quota*, or reports that it has *succeeded*, *deadlocked*, or *failed*. In the event of a deadlock, which is not cleared merely by the execution of goals delayed by *async/2*, the *wam_machine/4* goal suspends pending further activity in the Prolog engine bar persistent delayed goal execution. In the event of having run out of resources, it suspends on the tail of its resources stream *R*. Further resources can be given to the Prolog engine by appending a new *quota* element to the end of its *Resources* stream, allowing the Prolog computation to continue. Another primitive required is *write_and_wait(Term, Name, Term1)*. It executes atomically by writing *Term* to the file *Name* and suspending on all the variables in *Term1*. It succeeds if and when a variable in *Term1* is bound. These primitives can be used to define *prolog/1* in the CLP language as follows:

```
prolog(Goals) :- wam_machine(Name, resolve(Name), [quota|Cs], S),
                 write(Goals, Name),
                 manage(Goals, Name, Cs, S).
```

```
manage(Goals, Name, Cs, [used_quota|Ss]) :- write(Goals, Name),
                                             read(Goals, Name) |
                                             Cs = [quota|Cs1],
                                             manage(Goals, Name, Cs1, Ss).
manage(Goals, Name, Cs, [succeeded|Ss]) :- read(Goals, Name).
manage(Goals, Name, Cs, [deadlocked|Ss]) :- handle_deadlock(Goals, Name, Ss) |
                                             manage(Goals, Name, Cs, Ss) ;
manage(Goals, Name, Cs, [_|Ss])          :- write('$fail', Name).
```

```
handle_deadlock(Goals, Name, Ss) :- instantiated(Ss) | true;
handle_deadlock(Goals, Name, Ss) :- write_and_wait(Goals, Name, (Goals, Ss)),
                                     read(Goals, Name) |
                                     handle_deadlock(Goals, Name, Ss).
```

prolog/1 creates a new Prolog computation thread using *wam_machine/4*. It writes *Goals* to it, and then lets the concurrent goals for the relations *wam_machine/4* and *manage/4* continue the transient communication. Whenever the computation uses a quota, deadlocks, wakes, or succeeds, the CLP computation receives notification of this on the status stream of the *wam_machine/4* goal. The CLP computation then sends to the Prolog computation its current version of the Prolog goal, and reads the CLP computation's current version of the goal. *handle_deadlock/3* handles deadlocks on the Prolog computation. It waits on receiving a message on the *Status* stream or failing that on further instantiations of *Goals*, passing *Goals* back and forth with each instantiation until the deadlock clears. The primitive *instantiated/1* tests if its argument is bound *at the time of call*. The interface in Prolog is as follows:

```
resolve(Name) :- read(Goals, Name),
                 async(Name, talk(Goals, Name)),
                 call(Goals),
                 write(Goals, Name).
resolve(Name) :- write('$fail', Name),
```

resolve/1 reads in *Goals*, sets up a communication link with the CLP computation using *talk/2*, tries to satisfy *Goals*, and writes the final version of the goal *Goals* back to the CLP computation.

6. A Coupled Meta-Interpreter

A coupled meta-interpreter illustrates how expressive this approach is. The CLP language differs from Parlog in eschewing sequential conjunctions and modes, in input matching each head argument, and in sequencing head matching before guard evaluation. *meta_clp(Goals)* invokes a CLP meta-interpreter.

```
meta_clp((A, B)) :- true          | meta_clp(A), meta_clp(B).
meta_clp(A)      :- primitive(A) | call(A);
meta_clp(A)      :- true          | clauses(A, Cls), reduce(A, Cls, B), meta_clp(B).
```

```
reduce(Goal, [C|Cls], B1) :- test(Goal, C, B)      | B = B1.
reduce(Goal, [C|Cls], B1) :- reduce(Goal, Cls, B)  | B = B1.
reduce(Goal, [(C;_)|Cls], B1) :- test(Goal, C, B)  | B = B1;
reduce(Goal, [(_;C)|Cls], B1) :- reduce(Goal, [C|Cls], B) | B = B1.
```

```
test(Goal, Clause, B) :- match(Goal, Clause, Guard, Body) | B = Body, call(Guard).
```

```
match(Goal, C, G1, B1) :- melt(C, (H :- G | B)) | Goal => H, G = G1, B = B1.
```

meta_clp/1's first clause handles conjunctions. Its second clause evaluates primitives with *call(A)*. Its third clause handles committed choice resolution. Frozen clauses retrieved in *clauses/2*'s second list argument define the first argument relation with ";" operators joining adjacent clauses. These clauses are tested by *reduce/3*. *reduce/3*'s first clause examines whether the leading clause can reduce the goal. The second clause searches the next clause in parallel. The third clause examines the last in a group of clauses to be searched in parallel. Clause search only continues after all previous *reduce/3* clauses have failed to reduce the goal. *test/3* performs the head match before calling the guard. *match/4* uses the primitive *melt/2* to obtain a fresh melted clause copy before input matching the goal with its head using the one-way unification primitive *=>/2* which only binds or shares right hand argument variables [15].

```
meta_prolog(Goals) :- solve(Goals, Cut, V), ( V = true; ( V = fail, Cut = cut, !, fail ) ).
```

```
solve(!, cut, true).
solve(!, cut, fail) :- repeat.
solve((A,B), Cut, V) :- !, solve(A, Cut1, V1),
                       ( ( Cut1 = cut, V1 = fail, repeat); solve(B, Cut2, V2) ),
                       value(Cut1, V1, Cut2, V2, Cut, V).
solve(A, nocut, true) :- resolve(A).
```

```
resolve(G) :- primitive(G), !, call(G).
resolve(G) :- clause(G, Body), solve(Body, Cut, V),
              ( V = true; ( V = fail, Cut = cut, !, fail ) ).
```

```
value(cut, fail, _, _, cut, fail) :- !.
value(_, _, cut, V, cut, V)       :- !.
value(Cut, V, _, _, Cut, V).
```

meta_prolog(Goals) invokes a Prolog meta-interpreter which handles primitives and the cut. This call is expanded by two arguments to record the passing of cuts and whether *Goals* is true or fails. The first time a cut is met in trying to satisfy *Goals*, the first clause for *solve/3* makes this a *cut true* combination. Subsequent backtracking makes the

second clause turn this to a *cut fail* combination. These *cut fail* combinations are propagated across the goal level they originated on by the third clause for *solve/3*. Goals in the body after an earlier occurrence of a *cut fail* combination are ignored until the whole clause body has been traversed. The *cut fail* valuation cannot be undone by backtracking, once it has been created, because of the infinitely resatisfiable goal *repeat*. The second clause for *resolve/1* shows that if in the execution of a clause body, a *cut fail* combination is met, then the parent goal fails. However, the parent goal does not pass back details on why the goal failed. Thus the effect of a *cut fail* combination becomes simple goal failure at a higher level. *value/6* combines the cut status for goal conjunctions.

A coupled meta-interpreter can be adapted from the CLP and Prolog meta-interpreters given earlier.

```
meta_clp((A, B))    :- true           | meta_clp(A), meta_clp(B).
meta_clp(prolog(A)) :- nonvar(A)      | prolog(meta_prolog(A));
meta_clp(A)        :- primitive(A)   | call(A);
meta_clp(A)        :- clauses(A, [C|Cs]) | reduce(A, [C|Cs], B), meta_clp(B);
meta_clp(A)        :- true           | meta_prolog(A).
```

The meta-interpreter is invoked at the CLP side by *meta_clp(G)*. The new clauses for *meta_clp/1* work like the old except for the second and last clauses. The new second clause handles the transiently communicating interface to Prolog *prolog/1*. It uses the primitive *nonvar/1* to delay execution until the argument *A* is bound. The new last clause handles the atomic interface to Prolog. The last clause is invoked only if the goal argument of *meta_clp/1* is neither a conjunction, a primitive, *prolog(A)* or user defined, which makes *meta_prolog/1* by default a call over a definite clause relation.

The Prolog side assumes that invocations of the transiently communicating interface *clp/1*, which occur in the body of a rule, are re-written into a single call to *clp_call/2* at the start of the rule body and a call to *data/1* at the end as explained earlier. The meta-interpreter is adapted from the earlier meta-interpreter for Prolog by changing the clauses for *resolve/1* to the following

```
resolve(G) :- primitive(G), !, call(G).
resolve(G) :- clause(G, Body), !, body(Body, B), solve(B, Cut, V),
              ( V = true; ( V = fail, Cut = cut, !, fail ) ).
resolve(G) :- functor(G, F, N), functor(G1, F, N), \\+ clause(G1, _), meta_clp(G).

body((clp_call(A,X), B), B) :- !, clp_call(A, X).
body(B, B).
```

The meta-interpreter is invoked at the Prolog side by *meta_prolog(G)*. The old second clause for *resolve/1* is replaced by a new clause. It uses *body/2* to handle the transiently communicating interface to the CLP computation. The new

third clause handles the atomic interface to Prolog. Its body is only invoked if the argument to *resolve/1* is neither a primitive nor a user defined Prolog goal.

7. Multiple Solutions Predicates

Destructive assignment can be used to program eager and lazy multiple solutions interfaces, *set/3* and *subset/3*, to Prolog from Parlog on a Parlog-Prolog coupled system using an interface similar to *prolog/1* [5]. However, if destructive assignment is only used as an implementation technique and not made accessible to a user, then the user cannot use the same method to program variations on these multiple solutions predicates. He is forced to use *set/3* and *subset/3* to obtain multiple solutions. To avoid this weakness a different method is proposed for supporting eager and lazy multiple solutions calls from the CLP engine to Prolog. It can be adapted by users for other purposes without using destructive assignment. An eager all-solutions predicate *eager/3* like Parlog's *set/3* is defined in the CLP language as follows

```
eager(T, Goal, Sols) :- gensym(Name) | all(Name, T, Goal), solutions(Name, Sols).
```

```
solutions(Name, L) :- element(Name, A) | L = [A|M], solutions(Name, M);  
solutions(Name, L) :- true | L = [].
```

```
element(Name, A) :- defined(Name, 2) | S =.. [Name, A, B], retract(S), B == yes.
```

```
update_db(Name, A, B) :- true | S =.. [Name, A, B], assert(S).
```

eager/3 obtains a unique symbol *Name* from *gensym/1* and uses the Prolog relation *all/3* to compute each solution, collecting them using *solutions/2*. Solutions are passed from Prolog to the CLP engine's database using the relation *update_db/3* which appends them to the end of the relation *Name/2*. These solutions are progressively picked up from the database by *element(Name,A)*. *element/2* fails on the *no* labelled clause after the last solution has been picked up. The CLP primitive *defined/2* suspends until the CLP database is defined for *Name/2*, and then the new solution is retracted. *=../2* and *==/2* act like Prolog's similar primitives, but suspend if their arguments are insufficiently instantiated for them to succeed or fail. *all/3* is defined in Prolog as

```
all(Name, T, G) :- copy((T,G), (T1,G1)), every(Name, T1, G1).
```

```
every(Name, T, G) :- call(G), update_db(Name, T, yes), fail.  
every(Name, T, G) :- update_db(Name, _, no).
```

copy/2 ensures its arguments are copies. The solution template *T* and the goal *G* are copied to ensure that revoked

bindings are not passed to the CLP computation. Solutions are created by backtracking and passed to the CLP database using the atomic interface by invoking the CLP relation *update_db/3*. If the Prolog database contains the clauses below then the call *eager(X, person(X), L)* binds L to [aristotle, kant].

```
person(aristotle).
person(kant).
```

A lazy multiple solutions *lazy/3*, like Parlog's *subset/3*, can be defined in the CLP language. However, the *No Complex Term CLP Bindings* restriction prevents this predicate behaving like *subset/3*. The restriction means that the list of solutions L cannot be incrementally extended in the CLP engine and these new list extensions passed transiently to the Prolog computation for adding new solutions into. Instead each solution element is pre-specified as a structure (*Ready, Variable*), and the full length of the solution list is pre-given. When the next element of the solution list has its *Ready* variable bound to any atom, then *Variable* can be bound to a solution. If more solutions are wanted than can be supplied, the rest of the solution elements are bound lazily to *end*. Thus the call over the previous database

```
lazy(X, person(X), [(ok,A), (ok,B)]).
```

would bind A and B to *aristotle* and *kant*, and the extra element in

```
lazy(X, person(X), [(ok,A), (ok,B), (ok,C)]).
```

would be bound to *end*. *lazy/3* can be defined in the CLP engine as follows

```
lazy(T, G, L) :- gensym(Name) | prolog(some(Name, T, G, L)), more(Name, L).

more(Name, [(_,A)|M]) :- defined(Name, 1) | S =.. [Name, A], retract(S), more(Name, M).
more(Name, [(_,end)|_]) :- true | true.
more(Name, []) :- true | true.

revise_db(Name, A) :- true | S =.. [Name, A], assert(S).
```

where *some/4* is defined in Prolog as

```
some(Name, T, G, L) :- copy((T,G), (T1,G1)), assert(to(Name,0)), each(Name, T1, G1, L).
```

```
each(N, T, G, [(R,_)|L]) :- data(R), call(G), revise_db(N, T), next(N, L, 0).
```

```
each(N, _, _, [(_,end)|L]) :- end(L).
```

```
each(N, _, _, []).
```

```
next(N, [(R,_)|T], A) :- retract(to(N, A)), B is A + 1, assert(to(N, B)), data(R), !, fail.
```

```
next(N, [_|T], A) :- B is A + 1, next(N, T, B), !.
```

```
next(N, [], _).
```

```
end([(R,A)|L]) :- data(R), A = end, end(L).
```

```
end([]).
```

Unlike *eager/3* the lazy multiple solutions predicate *lazy/3* uses the transiently communicating interface *prolog/1* to allow transient communication of new solution requests. Otherwise the mechanism of passing of solutions is essentially the same. The goal *revise_db/2* plays the role of *update_db/3* of passing solutions to the CLP engine's database. *next/3* ensures that the Prolog engine suspends using the delaying primitive *data/1* on the signal element of the next empty solution structure until the signal element is bound by the CLP computation. *end/1* lazily binds solution elements of solution structures to *end* for solution requests which exceed what can be generated.

8. Conclusion

A deterministic coupling of Prolog and CLP resolution engines enables the use of trial bindings in a don't know non-deterministic search to be safely isolated and hidden within a Prolog sub-computation. This allows CLP resolution to sustain a system of concurrent communicating tasks without risk of uncommitted bindings rolling back the history of the system interaction in an anachronistic way. Four interfaces between coupled CLP and Prolog computations have been defined, and a simple non-shared memory implementation coupling scheme for single and multiple processor machines has been proposed, which does not interfere with the basic efficient mechanics of CLP and Prolog resolution engines. A concise CLP-Prolog meta-interpreter for all four interfaces was given. It helps prove the simplicity and power of the interfaces. The effect is to achieve a coupled logic programming system which can manage systems programming tasks on the CLP engine, and knowledge base querying on the Prolog engine. It can support multiple concurrent activities in each area at the same time. Concurrency is partly realised through the Unix operating system among the CLP engine and various *tightly* or *loosely* coupled Prolog processes. It is also partly realised through the concurrency handling mechanisms of the CLP engine.

Acknowledgements

This research was funded by Alvey project IKBS 90.

References

1. R.G. Babb(II), *Programming Parallel Processors*, Addison-Wesley Publishing Company, New York, (1988).
2. R. Bahgat and S. Gregory, *Pandora: Non-deterministic Parallel Logic Programming*, Department of Computing, Imperial College, London, (November 1988).
3. M. Carlsson, "Freeze, Indexing and Other Implementation Issues in the WAM," in *Proceedings of 4th International Conference on Logic Programming*, ed. J-L. Lassez, pp. 40-58, MIT Press, Melbourne, (May 1987).
4. K.L. Clark and S. Gregory, "Notes on the Implementation of Parlog," *Journal of Logic Programming*, **Vol. 2**, (1), pp. 17-42, (July 1985).
5. K.L. Clark and S. Gregory, "Parlog and Prolog United," in *Proceedings of 4th International Conference on Logic Programming*, ed. J-L. Lassez, pp. 927-961, MIT Press, Melbourne, (May 1987).
6. J.S. Conery, *Parallel execution of logic programs*, Kluwer Academic Publishers, Boston, Massachusetts, (1986).
7. H. Fujita, "FGHC Partial Evaluator as a General Purpose Parallel Compiler," ICOT Technical Report TR-386, Institute for New Generation Computer Technology, Tokyo, Japan, (May 1988).
8. S. Gregory, I.T. Foster, A. Burt, and G.A. Ringwood, "An Abstract Machine for the Implementation of Parlog on Uniprocessors," *New Generation Computing*, **Vol. 6**, (4), pp. 389-420, (1989).
9. S. Haridi and P. Brand, "Andorra Prolog: an integration of Prolog and committed choice languages," in *Proceedings of the International FGCS Conference*, Tokyo, (December 1988).
10. J.W. Lloyd and R.W. Topor, "A Basis for Deductive Database Systems II," *Journal of Logic Programming*, pp. 55-67, (1986).
11. Sun Microsystems, *System Services Overview*, Manual for Sun Microsystems, (May 1988).
12. L. Naish, "Parallelizing Nu-Prolog," in *Proceedings of the 5th International Conference/Symposium on Logic Programming*, ed. K.A. Bowen, pp. 1546-1564, MIT Press, Seattle, Washington, (August 1988).
13. E. Shapiro, "Concurrent Prolog: A Progress Report," Technical Report CS86-10, Department of Computer Science, The Weizmann Institute of Science, Rehovot, Israel, (April 1986).
14. E. Shapiro, "An Or-Parallel Execution Algorithm for Prolog and its FCP Implementation," in *Proceedings of 4th International Conference on Logic Programming*, ed. J-L. Lassez, pp. 311-337, MIT Press, Melbourne, (May 1987).

15. H. Taylor, *A Lingua Franca for Concurrent Logic Programming*, pp. 1-20, Computer Science Dept, Queen Mary College, London University, London, (January 1990).
16. H. Taylor, *Logical Exceptions*, pp. 1-19, Computer Science Dept, Queen Mary College, London University, London, (January 1990).
17. K. Ueda, "Making Exhaustive Search Programs Deterministic," in *Proceedings of the 3rd International Logic Programming Conference*, ed. E. Shapiro, pp. 270-282, Springer-Verlag, London, (July 1986).
18. L. Vieille, "A Database-Complete Proof Procedure Based on SLD-Resolution," in *Proceedings of 4th International Conference on Logic Programming*, ed. J-L. Lassez, pp. 74-103, MIT Press, Melbourne, (May 1987).
19. R. Yang, "P-Prolog: A Parallel Logic Programming Language," PhD thesis, pp. 1-138, Keio University published by World Scientific, Singapore, (1987).