Compiling cp(↓, | ,&) on top of Prolog

Vijay A. Saraswat

October 1987
CMU-CS-87-174

Abstract

In this paper we present an implementation for the concurrent logic programming language cp(↓ , | ,&). The implementation compiles such programs into **Prolog** programs, which may further be compiled and run on **Prolog** systems. The design differs significantly from [Ueda and Chikayama, 1985].

We present the implementation in a series of steps. We start with a design that compiles cp(↓, | ,&) programs into a cp(f, | ,&; o), which corresponds closely to **Prolog**, enhanced with a 'geler' (or `freeze`) capability, ([Boizumault, 1986]). The design uses a concept of *modes* to partition the clauses for a predicate into equivalence classes such that all the clauses in one class have identical suspension conditions. Multiple modes are examined 'simultaneously' by invoking a 'mode-goal' for each mode for that goal, and arranging for distributed commitment of these mode-goals using mutual and single exclusion. Because of the static nature of cp's '↓'-annotation (as opposed to **Concurrent Prolog**'s '?'), a number of important optimisations are possible.

Most implementations of **Prolog**, however, do not have a `freeze` capability. We present a meta-interpreter for cp(f, | ,&; o). By adding a 'suspension queue', we can obtain an interpreter for cp(f, | ,&; o) in cp(| ,&; o). The code generated by the cp(↓, | ,&) compiler may be thought of as being obtained by partially evaluating the output of the cp(↓, | ,&) to cp(f, | ,&; o) translator with respect to this interpreter to obtain a cp(| ,&; o) program. cp(| ,&; o) programs may be trivially implemented in **Prolog**. Finally, we present a number of important optimisations for multi-mode predicates, and compare the performance of this compiler with [Ueda and Chikayama, 1985].

This research was sponsored in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4976, monitored by the Air Force Avionics Laboratory Under Contract F33615-84-K-1520.

Contents

1 Introduction

In this paper, we present an implementation of the language cp(↓, | ,&), which is based on the concurrent and non-deterministic interpretation of definite clause programs.

The cp family of programming languages originated in an attempt to understand Shapiro's **Concurrent Prolog**, and to determine whether the two major language proposals contained therein — synchronisation through read-only variables and determinate communication via don't care commitment – were *necessary* for a useful concurrent interpretation of definite clause programs. In [Saraswat, 1986], I showed that there were some problems with the definition of read-only annotation, particularly in the context of a language with 'full' guards, and that a much simpler static *wait* annotation ('↓') was instead adequate for sychronisation. I also showed that, through the notion of *don't know commitment* ('&') there was a conceptually very straightforward way to exploit the choice non-determinism inherent in the programming language **Prolog** in particular and definite clause logic programming in general. In this framework, the *don't care commitment* (' | ') so central to the so-called 'committed choice concurrent logic programming languages' (**Concurrent Prolog, Parlog, GHC**) reappears with the same status that the 'cut' has in **Prolog**: it is an important aid to the system in detecting determinate computations, which may be executed more efficiently (choice points may be reclaimed,or not created in the first place.) But its use is not essential in every clause! Indeed, the relationship between cp(↓, | ,&) and **Prolog** is very close: the latter may be regarded as a sequential (and impure — in a technical sense made precise in [Saraswat, 1987d]) approximation of the former.

The formal semantics of cp(↓, | ,&) has.been studied in [Saraswat, 1985] and [Saraswat, 1987b], and several fundamental programming techniques presented in papers such as [Saraswat, 1987, Forthcoming] and [Saraswat, 1987a]. [Saraswat, 1987c] presents cp(↓, | ,&) as a programming language that strongly supports the paradigm of concurrent, controllable constraint systems. Indeed, by exploiting concurrent forward checking, local propagation and dynamic variable reordering, many of the problems of chronological backtracking that plague the use of **Prolog** as a constraint language can be avoided, without having to pay the computational overhead of such schemes as dependency directed backtracking.

In this paper we discuss an efficient implementation of the language via compilation into **Prolog**.

A number of ways of implementing concurrent logic programming (CLP) languages have been investigated. In [Shapiro, 1983], Shapiro presents an interpreter for (a subset of) **Concurrent Prolog** on top of **Prolog** (we discuss it briefly in the next section). While rather simple and quite portable, it was unacceptably slow, sometimes about two orders of magnitude slower than the underlying **Prolog** system. To remedy the situation, a compiler from **Flat Concurrent Prolog** to **Prolog**, with much better performance is presented in ([Ueda and Chikayama, 1985]). Since the compiler and the run-time system were both written in **Prolog**, the advantages of portability were not lost. Similar efforts have been reported for the programming language **Parlog**.

More serious implementations have been undertaken, including native mode code generators from abstract machines designed to implement determinate computations efficiently ([Silverman, 1986]. See also [Kliger, 1987].) However, the scale of effort involved here is of the order of man-years, and this work is only now coming to fruition. Note that this work does not directly support non-deterministic ('backtrackable') choice: preliminary work on supporting even 'pure' non-determinism (much less a language like cp(↓, | ,&)) on top of these implementations ([Ueda, 1986], [Shapiro, 1987]) seems only mildly promising, at best.

In this paper, we choose to implement cp(↓, | ,&) by compiling into **Prolog**. In the long run, as with other workers in the CLP field, our goal is to develop real implementations for the languages we are concerned about, and to that extent, this work must be seen as preliminary in nature to work relating to the design of abstract machines and processor architectures more directly supporting the cp model of computation. In the short term, however, we feel that it makes sense to take advantage of the considerable research work that has been done, and is being done, related to the very efficient implementation of **Prolog** on uni- and multi-processors. The last point takes on added significance, in the present case. For cp(↓, | ,&), through don't know choice, Or-parallelism plays a signficiant role. Recently efficient Or-parallel implementations for **Prolog** have been devised ([Warren, 1987]): in so far as our implementation produces **Prolog** programs which create and use choice points, the techniques employed in their implementations are directly applicable.

Moreover, developing an implementation on top of **Prolog** usually involves a few man weeks of effort. This is important because we expect to use the system, and various tools such as partial evaluators, to experiment with the design of concurrent constraint based programming languages more directly tailored to expressing user-constraints. Finally, building the system on top of **Prolog** makes it immediately accessible to the logic programming community at large, and establishes links with the work of those people who have been trying to (directly) use **Prolog** for constraint-based computations.

We have sought to provide an implementation in which as much work as possible is done by the underlying **Prolog** implementation, and about whose correctness we can make convincing arguments. We have also been concerned with structuring the implementation in such a way that the natural restriction of the compilation scheme to cp(↓, |) programs produces efficient code. In other words, if a particular program does not use don't know commitment, it does not have to pay any implementation overhead. These efficient implementations immediately provide reasonably fast implementations of the concurrent logic programming language in question.

The rest of this paper is as follows. For the sake of completeness, we present a brief discussion of the syntax and semantics of the language cp(↓, | ,&). We discuss related implementation schemes, and their unsuitability for our language. The design differs significantly from [Ueda and Chikayama, 1985]. It distinguishes between deadlock and failure, has different mechanisms for suspension-handling, uses unbounded depth-first scheduling, implements 'don't know non-determinism' and also a well-defined restricted subset of full guards called 'wait-free' guards, and has very different (and significantly better) performance characteristics.

We present the implementation in a series of steps. We start with a design that compiles cp(↓, | ,&) programs into cp(f, | ,&,o) programs. The design uses a concept of *modes* to partition the clauses for a predicate into equivalence classes such that all the clauses in one class have identical suspension conditions. For a given goal, the suspension condition of a mode is checked at most once very time the goal is scheduled. Multiple modes are examined 'simultaneously' by invoking a 'mode-goal' for each mode for that goal, and arranging for distributed commitment of these mode-goals using mutual and single exclusion techniques. An important benefit of this implementation technique is that 'wait-free' cp programs (i.e. cp(| ,&) programs) run at the speed of the **Prolog** program with identical clauses. Another important optimization is that since single mode predicates can be detected at compile time (a property of cp not shared by 'dynamic annotation' languages such as **Concurrent Prolog**), code can be generated that avoids the overhead of distributed commitment. Next, we show that with the additional assumption that guards are wait-free, cp(f, | ,&,o) programs may be trivially translated into a **Prolog** (with 'cut') enhanced with a 'geler' (or `freeze`) capability, ([Boizumault, 1986], [Carlsson, 1987]).

The correctness of the design given above, however, depends upon some delicate properties of `freeze` that do not obtain in some (most?) current **Prolog** implementations ([Carlsson, 1987]) that support `freeze`. A suitably designed `freeze` capability has the potential for offering non-busy-wait implementations of cp(↓, | ,&): a suspended goal is not scheduled for execution unless its suspension conditions are satisfied. We discuss the drawbacks in the implementations of `freeze` discussed above, and offer alternative designs for `freeze`.

Most implementations of **Prolog**, however, do not have a `freeze` capability. We next provide a simple meta-interpreter for cp(f, | ,&; o), and use it to implement an interpreter for cp(f, | ,&,o) in cp(| ,&; o). This interpreter uses a global 'suspension queue' to implement the `freeze` predicate. We observe that, in principle, partially evaluating the **Prolog**-with-`freeze` output of the cp compiler with respect to this enhanced meta-interpreter (for **Prolog**) should yield a **Prolog** program which directly implements the original cp program. In this program the global suspension queue, and deadlock flag are passed around as extra arguments to every (object-level) call instead of being maintained in the meta-interpreter state. Our compiler may be thought of as implementing such a translation into **Prolog** code.

Quite importantly, a number of important optimizations are possible when compiling directly into **Prolog** (and bypassing `freeze`). Instead of spawning goals for all the Or-candidates of a cp goal, only one goal may be spawned, and a list of modes with respect to which the goal is to be evaluated may be passed to it. This results in a considerable saving in runtime and heap space. Another important saving results from treating the case of the first invocation of a multi-mode predicate separately from subsequent invocations. Instead of passing a fully constructed modelist as an argument to the first call, the list is incrementally constructed as various modes are examined, and then too, only if necessary. Another important optimization concerns multi-mode goals that are scheduled and suspended frequently: their modelists are examined in such a way that a new modelist (for the goal to be suspended) is not created unless the previous one is to be modified. This results in considerable space savings for some programs.

We present examples of code generated by the compiler for some representative programs, and present statistics for versions of the compiler that support different optimisations. We compare the performance of our compiler with that reported in [Ueda and Chikayama, 1985].

Finally, we discuss opportunitics for further work. It seems clear that in some cases identical suspension tests in different modes can bc avoided. Designing a general scheme for detecting such duplications and avoiding as many as possible should be interesting. It also seems quite feasible that these ideas can be extended to design an abstract architecture that supports cp(↓, | ,&), in the same way the WAM supports **Prolog**.

Before presenting the implementation scheme I would like to offer a word of caution. Having an implementation of a concurrent and non-deterministic programming language early in the course of its development is not without its problems. It can sometimes encourage the adoption of programs as 'correct' because their execution on a given implementation always seems to yield correct answers. All that an implementation can guarantee is that if it is correct, then correct programs will run correctly; however it may also be that faulty progams in fact seem to behave correctly because of some qirks in the scheduler! Correctness of programs has to be proven with respect to an underlying formal semantics and, if there is one, a correctness calculus.

Finally, it is worth noting that the speed of a sequential implementation may not be a very good indicator of the speed of a parallel implementation. The analysis of programs should be done with respect to the model of computation, and any information that the language implementer may provide about the effficiency of implementation of the abstract model on the current architecture. The current state of the art in writing performance efficient concurrent logic programs is not well developed.

2 Background and related work

2.1 The language cp(↓, | ,&)

The cp family of languages is based on the concurrent interpretation of definite clause programs. The discussion in the rest of the section is based on [Saraswat, 1987b], though the syntax here is newer.

2.1.1 Nomenclature for languages in the cp family

The name of a language in the cp family consists of the symbol 'cp' prefixed with one or more *qualifying* letters and suffixed with a *delimiter* separated list of *indicators*. For the purposes of this paper, the only qualifying letter discussed is 'f': when present the language is said to be *flat*. The delimiters are: '`;`' (And-sequentiality) and '`;;`' (Or-sequentiality). When neither are present in the language, or the name has at least two more indicators than delimiters, ',' may be used to delimit the indicators. The indicators are: '↓' (the *wait* annotation), 'f' (the *freeze* construct), '|' (*don't care commitment*), '&' (*don't know commitment*), and 'o' (the *otherwise* construct). When more than one control construct is present, it is listed in the name of the language in the above order.

The only other restrictions on constructing the name of a language (and hence on the combination of control constructs allowed in the languages!) are: the name must have at least one commitment operation, cannot contain both f and '↓', and cannot contain both o and ';;'. In the last two cases, the first is a special case (more restrictive version) of the second, as we shall see.

We have chosen this systematic nomenclature to immediately make evident what are the control constructs (deviations from pure definite clause programming) being discussed in the language at hand. (We do not consider And- and Or-parallelism to be deviations from pure definite clause programming.) This is particularly important in this paper because we shall present a number of translations from one language to another. However, we shall not attempt to counsel the reader on how to pronounce these names.

2.1.2 Syntax

We present the syntax of programs in the language incrementally. We assume as primitive the ideas of *variables*, *terms* and *atoms*. We define the syntactic categories: `Program`, `Clause`, `Head` and `GoalSystem`. Each of these syntactic categories are ranged over by the (possibly subscipted) meta-variables `P`, `c`, `h`, `g`. Atoms for user-defined predicates are ranged over by the (possibly subscipted) meta-variable `a`. Atoms constructed from a set of pre-defined ('built-in') predicates are ranged over by the (possibly subscripted) meta-variables `b`.

All languages share the following production rules defining the categories above:

`P ::== c.`	`c ::== c.c`	`h ::== a`
`g ::== a`	`g ::== b`	`g ::== g,g`

Depending upon which commitment operation(s) are available in the language, to the table above add the production(s):

`c ::== h ← g & g`
`c ::== h ← g	g`

If the language allows '↓'-annotations, add the rule:

`h ::== ā`

where $\bar{a}$ ranges over the syntactic category `AnnAtom` discussed below.

If it has And-sequentiality or Or-sequentiality, add the appropriate rule from:

```
g ::== g ; g
c ::== c ;; c
```

If the language name contains 'f' appears in the name of the language, then the set of built-in predicates includes the predicates `freeze/1`, `freeze/2` and `freeze/3`, discussed below. If the language name contains 'o', then the set of built-in predicates includes the predicate `otherwise/0`. In addition, programs must satisfy the syntactic restiction that an `otherwise/0` goal can occur only in the guard of a clause, and then as the first goal in the guard. If the symbol in the language name is prefixed with an 'f', then only goals for built-in predicates may appear in the guards of clauses.

An *annotated atom* may contain terms which are *annotated*. There are two types of annotations: a *wait* annotation '↓', and an *equate* annotation '↓ T', where T is some constant, called the *marker* for the annotation. A term is annotated by sufffixing the appropriate annotation to the term. A term of the form S↓ is said to be '↓'-annotated. and of the form S ↓ T is said to be T-annotated. When the distinction is not relevant, we shall simply speak of annotated terms.

Finally we discuss some shorthands. The set of built-ins contains the special predicate `true/0`. If the language provides the '|'-commitment operation, then clauses on the left may be notated more simply by the syntax on the right:

Verbose syntax:	*Shorthand:*
`c ← true	g.`	`c ← g.`
`c ← true.`	`c.`

(The second shorthand rule is applicable only after the first has been used once!)

2.2 Informal semantics

We now discuss informally the semantics of these control constructs. The discussion assumes a notion of '↓'-*unification*, i.e. unification in the presence of annotations. This is discussed in detail below: at this point, suffice to say that '↓'-unifying a goal against the head of a clause may result in either success, or failure or *suspension*.

In overview, computation is initiated by the presentation of a *query* $Q \equiv a_0 \ldots a_{n-1}$. Each goal a_i will try to find a proof by '↓'-unifying against the head of a clause, and finding proofs for the goals in the guard and body of that clause. Unification results in *bindings* for the variables in the goal, which are *communicated* (applied) to the other sibling goals of the committing goal. If the goal commits using a don't care commit then the bindings it commits are irrevocable, and are directly applied to the sibling goals. If the goal commits using a don't know commit, then the choice points for the goal are preserved, the sibling goals are copied, and computation forks into two branches. In one branch the bindings being committed by the goal are applied to the sibling goals; in the other they are not. Computation succeeds when any branch finds a solution; it fails when all branches terminate in failure.

In more detail, each goal in Q tries to *reduce* itself to a goal-system by '↓'-unifying against the head of a clause, say, ($C \equiv a \leftarrow g_1 \;\%\; g_2.$). If $\theta = \mathbf{mgu}_{\downarrow}(a_i, a)$ $(i < n)$ exists, then the goal-system $\theta(g_1)$ is invoked as a *guard system.* Each a_i may have more than one guard systems (called Or-siblings of each other) executing concurrently. If the head-unifier produces more than one substitution, then an a_i may actually have more than one guard -system in the *same* clause: each guard system corresponds to one substitution. Such 'internal' Or-siblings are to be thought of exactly as if there were multiple copies of each clause, with one guard system in each clause. In what follows, then, we will discuss only the case in which there is one guard system per clause.

Each of the guard systems is a goal-system containing goals, which, in turn, may invoke other clauses (with guard systems) and hence a tree of goals can be built up in the guard. Bindings committed by goals within the `GoalSystem` accumulate at the boundaries: they are not propagated further till the goal-system is completely solved. Whatever bindings a guard goal commits during the process of its execution are, therefore, not visible to its environment (sibling goals of the atom invoking the guard, or body goals in the same clause). They are visible only to other sibling goals within the guard.

Assume that some guard system for some goal a_i has been solved successfully with answer substitution θ. In the above example, if clause C has an empty guard system, it immediately terminates with answer substitution θ =the unifier of a_i and a, the head of clause C. If more than one of the guard systems terminate successfully at the 'same time', one of them is chosen to *commit*, by some mutual exclusion algorithm.

Commitment involves three operations: the *atomic publication* of the answer bindings, action on other Or-siblings and promotion of body goals.

Atomic publication of bindings means that the bindings are (conceptually) instantaneously applied to all the goals in the body of the clause and to *all* And-sibling goals (and their Or-subsystems) of the committing goal a_i. (If some And-sibling goal has an Or-subsystem which already has bindings for variables that are incompatible with the bindings being published, the sub-system *fails*.)

Together with atomic publication, both the commit operations also cause the goals in the body goal-system to be executed as And-siblings of the goals that were the siblings of the committing goal (i.e. their uncles).

The commit operators differ in the actions they take with respect to other Or-siblings of the committing guard system, either in the same clause or in other clauses. The don't care commit *kills* them all. The don't know commit allows Or-siblings to keep on computing, in effect allowing a goal to commit *multiple* bindings. Each of these bindings is, committed to a *different* copy of the rest of the goals. Thus, if a goal '&'-commits bindings θ, all the sibling goals in the GoalSystem are split into two at commit time.

Computation succeeds when any branch finds a solution; it fails when all branches terminate in failure.

We now discuss some crucial aspects of the operational semantics in more detail.

2.2.1 '$\downarrow$'-unification

Informally, $\downarrow$-annotations in the head force the suspension of a goal until all the terms in the goal corresponding to $\downarrow$-annotated terms in the head are instantiated, and all the terms in the goal corresponding to the '$\downarrow$ m'-annotated terms in the head are equated, for every marker m. '$\downarrow$'-unification then succeeds iff normal unification of the goal and the (unannotated) head succeeds.

We now define the blocking condition for a clause head more precisely. First, we review some terminology, following [Courcelle, 1983].[1]

Let N denote the set of non-negative integers, N_+ the set of positive integers, and N_+^* the set of finite sequences over positive integers (including the empty sequence, ϵ). We indicate concatenation of finite sequences by juxtaposition. A *ranked alphabet* is a pair (F, ρ), where F is a possibly infinite collection of (function) symbols, and $\rho : F \rightarrow N$ is a maping that defines the arity (rank) of symbols in F. We consider sets F where the symbols of arity 0 are partitioned into *variables* and *constants*. A *tree over a ranked alphabet* (briefly: tree) is a partial mapping $t : N_+^* \rightarrow F$, such that

[1]The exposition presented here is a slightly modified version of the presentation in [Saraswat, 1986].

- its domain is prefix-closed and non-empty,
- if $\alpha \in N_+^*, i,j \in N_+, 1 \leq i \leq j$ and $\alpha j \in$ **dom**(t), then $\alpha i \in$ **dom**(t), and,
- if $t(\alpha) = f$, and f is of arity $k \geq 0$, then for $i \in N_+$, $\alpha i \in$ **dom**(t) if and only if $1 \leq i \leq k$.

We use the notation t/α for *the subtree of t issued from node* α, i.e. the tree $t' = \lambda\beta \in N_+^*.t(\alpha\beta)$.

Two trees t_1 and t_2 are said to be identical (notated $t_1 = t_2$) if their domains are identical, and for each path in their domain, their values are identical.

A *blocking condition* is a pair $\langle P, Q\rangle$, where P is a finite set of paths and Q a finite set of finite sets of paths. For a given clause, with head T, the blocking condition is the $\langle P, Q\rangle$ defined as follows. P is the prefix-closure of the set of all paths α in the domain of T such that $T(\alpha)$ is '$\downarrow$'-annotated, or a son of $T(\alpha)$ is '$\downarrow$ m'-annotated, for some marker m. Q is a set, indexed by the set M of all the markers occurring in T, such that Q_m is the set of all those paths α in the domain of T such that $T(\alpha)$ is '$\downarrow$ m'-annotated, for all $m \in M$.

For a given goal G and blocking condition $\langle P, Q\rangle$, the blocking condition *holds* if

- $P \subseteq$ **dom**(G) and if for all $\alpha \in P$, $G(\alpha)$ is a non-variable, and,
- for all elements $\{\alpha_1, \ldots \alpha_k\} \in Q$, $G/\alpha_1 = G/\alpha_2 = \ldots = G/\alpha_k$.

We will refer to thid definition when discussing how to generate code to implement '$\downarrow$'-unification.

2.2.2 Wait-free guards

Full guards are difficult to implement correctly and efficiently. For the purposes of this paper, we shall confine our attention to a subset of user-definable guards called *wait-free* guards.

To prepare for their definition, we define '*global variables*' to be variables that occur in the current resolvent. Now a clause of the form

```
Head  ←  Guard % Body.
```

is said to be wait-free iff the following condition holds:

> *Execution of even a single branch of* `Guard` *should not suspend on global variable(s) being instantiated or equated.*

The condition is simple to state, but the reader is urged to comprehend it fully. Some examples follow.

Note that execution of `Guard` may produce multiple branches — the goals appearing in `Guard` may be for predicates which have `&`-clauses. The definition above states that none of these branches may suspend on global variables. Note also that execution of `Guard` is not disallowed from producing bindings for global variables (cp is a 'multiple environment' family of languages).

This condition was chosen for two reasons: expressiveness and ease of implementation. Most examples where usage of non-flat guards seems to be indicated are covered by this restriction. For instance, I have found guards to be convenient to check some complicated predicate on user-defined data-structures, before commitment. The above condition will allow such guards as long as the head checks for whatever instantiatedness requirements the guard may have on the data-structure. [2]

[2]There is one important and useful piece of functionality that such guards cannot provide: *disjunctive* waiting for global variables, as done, for example, in the head of merge/3 clauses.

Wait-free guards are rather straightforward to implement on sequential machines. The goals in the guard are simply executed as a new (recursive) query in the current environment. Any deadlock that occurs in the execution of this query is presumed to not be resolvable even if the sibling goals of the invoking goal are scheduled for execution. Hence it is correct to treat this deadlock as failure, and cause it to trigger the search for alternative computations (i.e. backtracking) to find a solution for the guard. The sibling goals of the invoking goal are not scheduled for execution until the guard is completely solved (either successfully or unsuccessfully).

Example 2.1 (Wait-free guards) *As an illustration, consider the following program.*

```
producer(a).
producer(b).
consumer(X↓).
poss_producer(X) ←  consumer(X).
poss_producer(a).
n(X) ←  p(X).

q(X) ←  poss_producer(X) | true.
q(X) ←  poss_producer(X).
t(X) ←  q(X), producer(X) | true.
r(X↓) ←  poss_producer(X) | true.
s(X) ←  consumer(Y) | true.
```

The clauses for `producer/1, n/1 and poss_producer/1` *are (trivially) wait-free. However the first clause for* `q/1` *is not wait-free: execution of a query* `q(Y)` *may result in the goal* `consumer(X)` *suspending on global variable* `X`. *On the other hand the clause for* `t/1` is *wait-free. Even though* `q(X)` *occurs in the guard, and* `X` *occurs in the head of the clause,* `X` *also occurs in another goal* `producer(X)` *in the guard, execution of which will always generate a binding for* `X`.

The clause for `r/1` *is wait-free because* `X` *is guaranteed to be instantiated before the guard (which could potentially wait on* `X`*) is invoked. The clause for* `s/1` *is wait-free because its guard can suspend (in fact, deadlock!), but not on a global variable.*

2.2.3 Or-sequencing

The operational interpretation of a clause group `c1;;c2` is as follows. For any goal, clauses in `c2` are not considered as candidates until the clause group `c1` is determined not to contain even a potential candidate clause.

This definition has the following implications. If all the clauses in `c1` fail to commit, then the clauses in `c2` may be considered as candidates. If even a single clause in `c1` blocks either on head-satisfaction or guard-execution, and no clause in `c1` successfully commits, clauses in `c2` are not considered as candidates (because the clause that blocks is a potential candidate) until such time as (if ever) the clause unblocks and either fails or commits.

If a '`|`'-clause in `c1` successfully commits, then the clauses in `c2` are never tried. If a '&'-clause in `c1` successfully commits, then the clauses in `c2` are retained in the uncommited branch of the split, together with the remaining clauses in `c1`. As may be expected, in this branch the clauses in `c2` may be considered as candidates only when there are no remaining clauses in `c1`, or all of them are shown to be non-candidates.

otherwise. A special useage of Or-sequentiality deserves attention, and a special syntax. If all the clauses for a predicate can be written in the form $c_1;;c_2$, where c_1 and c_2 are Or-parallel clause groups, then the clauses c_2 may alternatively be called the otherwise clauses. Explicit useage of the ';;' operator may be avoided with the following syntax. The program may be notated as the Or-parallel combination $c_1.c_2'$, where c_2' is the same as c_2 except that the first goal in the guard of each clause in c_2 is a call to the special 'predicate' otherwise. This syntactic sugar allows the 'otherwise' clauses to be scattered in the body of the program.

2.2.4 Freeze

We introduce a special predicate freeze as an alternate suspension mechanism the better to introduce a smooth translation from cp(↓, | ,&) to **Prolog**. The predicate is based on a well-known idea in logic programming, apparently due to Colmerauer. See [Cohen, 1985,Boizumault, 1986,Carlsson, 1987] etc for details.

The predicate comes in three flavors: freeze/1, freeze/2 and freeze/3. A goal of the form freeze(A, Goal) suspends until such time as A is instantiated. It then reduces to the goal Goal. Such a predicate may be defined in cp(↓, | ,&) quite simply:

```
freeze(A↓, Goal)  ←  Goal.
```

A goal of the form freeze(A, B, Goal) suspends until such time as A has been equated to B. It then reduces to the goal Goal.

```
freeze(A↓ a, B↓ a, Goal) ←  Goal.
```

Finally, a goal of the form freeze(A) behaves as follows. The goal expects its argument to be a list of freeze/2 or freeze/3 data-structures. If one of the members of the list is of the form freeze(A, Goal) then the goal may, in one step, irrevocably reduce to the goal Goal after A is instantiated. Similarly, if one of the members of the list is of the form freeze(A, B, Goal), then the goal may, in one step irrevocably reduce to the goal Goal after A and B are equated. Such a predicate may be defined in cp(↓, | ;&) as follows. (Recall that cp(↓, | ;&) programs may be directly translated into cp(↓, | ,&).)

```
freeze(L) ←  freeze(1, M, L).
freeze(K, M, Null↓).
freeze(K, M, [freeze(A,Goal)|R]↓) ←
     (wait(A, M); commit(K, M, Goal)),
     freeze(M, K+1, R).

wait(Wait, M↓).
wait(A↓, M).
wait(A↓ a, B ↓ a, M).
  commit(K, K, Goal) ←  Goal.
commit(K, M↓, Goal) ←  M ≠ K |true.
```

We now turn to a discussion of some other implementation schemes for CLP languages.

2.3 Shapiro's Concurrent Prolog interpreter

In [Shapiro, 1983], Shapiro presents an interpreter for **Concurrent Prolog** programs. The interpreter accepts as input a database of **Concurrent Prolog** clauses. It maintains two data-structures: a queue of processes yet to be executed, and a deadlock indicator. When invoked with a sequence of goals, the interpreter enters them into the queue according to a scheduling policy, clears the deadlock indicator, and appends a `marker` goal at the end of the queue. Subsequently, on each iteration, the interpreter dequeues a process, and attempts to reduce it using the database of input **Concurrent Prolog** clauses. If it succeeds, the deadlock indicator is set, and the new goals, if any, added to the queue according to the scheduling policy. If it fails, the goal is added to the end of the queue. When the `marker` goal is encountered, the interpreter checks to see if the current queue is empty. If it is, the interpreter succeeds, and halts. If it isn't, then if the deadlock flag is still cleared, then it means that all the processes in the queue have been examined once, and none can reduce. The interpreter declares deadlock and fails. If the queue is not empty, and the deadlock flag is set, then a fresh marker is enqueued at the end of the current queue, and the deadlock flag cleared.

The structure of this interpreter seems basic to sequential implementations of concurrent logic programming languages. Dimensions along which implementations may vary is the type of scheduling policy used (discussed in detail below), and the extent to which part of the queue of suspended processes can be maintained within the call-stack of the underlying implementation language, thereby improving performance.

In Section 4.1, we present an interpreter for cp(f, | ,&,o), written in cp(| ,&; o), similar in spirit to this interpreter (though of course the interpreter does not have to be concerned with read-only unification).

2.3.1 Scheduling policy

What scheduling policy to use for the sequential implementation of CLP languages has been a source of some confusion in the literature. To my knowledge, neither **Concurrent Prolog** nor **GHC** nor **Parlog** initially made a commitment to And-fairness at the language level.[3] It is our contention that unless the language *semantics* specifies And-fairness, it is rather meaningless for a specific implementation to attempt to provide some form of fairness (typically, by using either breadth-first scheduling or 'bounded depth-first scheduling'). Usually, attempting to implement fairness involves some overhead: if nothing else, then in keeping count of how big a slice of processor time a particular process has already consumed. And, if the language does not specify fairness, but a program happens to terminate for a specific scheduler that attempts some form of fairness, it still says nothing about *the program*: the program may still not terminate on some other implementation perhaps employing a slightly different scheduling algorithm. When attempting to reason about the behaviour of a program, and when attempting to prove properties such as termination, one can only make those assumptions about the implementation which are codified in the definition of the language semantics.

cp(↓, | ,&) is not specified to be And-fair. Naturally, this means that if a system of processes contains even a single unconstrained producer, the system of processes is not guaranteed to terminate. But usually it is rather straightforward to harness such producers by means of dataflow synchronisation, and hence, as a practical matter this is not an issue. Or, to put another perspective on this issue, rather than the system providing some fixed scheduling policy such as '100-bounded depth-first scheduling', we feel that the user usually knows enough about the data-flow in his problem to pick and choose what kind of scheduling policy is appropriate in particular parts of his program, and to embed the scheduling policy

[3]Though, as we point out in [Saraswat, 1987e], it is rather difficult to make sense of GHC's suspension rules without assuming a notion of And-fairness.

directly in his source-level code by using data-flow synchronisation (e.g. bounded buffers). We feel that if such explicit control of resources eventually proves to be too cumbersome for the end-user, then fairness can be supported adequately in a 'higher-level' language on top of cp(↓,|,&) by providing a suitable meta-interpreter, and a partial evaluator for cp.

For these reasons, we only consider depth-first scheduling in this paper.

2.4 Ueda and Chikayama's Concurrent Prolog to Prolog compiler

This implementation ([Ueda and Chikayama, 1985]) may, in retrospect, be viewed as partially evaluating Shapiro's interpreter (a **Prolog** program), assuming a bounded depth-first scheduling policy, and giving it a fixed **Concurrent Prolog** program as input.

The global queue of suspended goals (also called 'continuation') and the deadlock flag are now passed to every goal in the system, instead of being state maintained by the interpreter. Hence corresponding to every **Concurrent Prolog** predicate `p/n`, there is a **Prolog** predicate `p/(n+5)`, where the 5 additional arguments are, in order, a counter specifying the current bound `B`, the head (`H`) and tail (`T`) of the suspension queue, the deadlock flag (`D`), and the maximum value of the counter (`Bmax`). There is one **Prolog** clause for every **Concurrent Prolog** clause in the original program; in addition, the **Prolog** predicate could have a prelude (one extra clause) and always has a postlude (another extra clause), both of which are discussed below.

Each **Concurrent Prolog** clause of the form

```
Head  ←  Guard | Body.
```

is translated into a **Prolog** clause of the form

```
(receiving arguments)  ←
     (Head unification),
     (bound check),
     (executing Guard), !,
     (decrementing bound),
     (scheduling Body).
```

The 'head unification' part has to call an explicit unification mechanism, because **Concurrent Prolog's** read-only annotation is implemented as a **Prolog** functor. This is a source of some (perhaps unavoidable) complexity and inefficiency in the system, and necessitates 'extra-linguistic' features, such as modes (discussed below), for obtaining performance.

The 'scheduling Body' portion of the clause is responsible for deciding what to do next. When there are no body goals in the clause, the first goal in the continuation is called. When there is exactly one goal in the body, the goal is executed, after being given the current continuation. When there are two or more goals in the body, the first one is called with a continuation that now has additional goals corresponding to the rest of the body goals.

The postlude clause simply requeues the current goal (unconditionally!) on the suspension queue, and calls the first goal in the continuation. This clause is invoked whenever the other clauses for the goal fail, before the cut in their body is crossed. Failure can occur either because of actual failure to unify, or because the suspension constraints were not satisfied. This means that even if *a goal really fails* (i.e. head unification and guard execution fails for all clauses) *it will still be enqueued on the suspension* *queue*. It also means that all guard executions — even if they have been executed and determined to fail based upon the current bindings — will be retried each time the goal is rescheduled, and this failure will be rediscovered each time. This could lead to considerable performance problems for some kinds of programs, in particular, those with 'non-flat' guards.

This problem (rescheduling failed goals and reexecuting guard systems which are bound to fail[4]) cannot be fixed in this framework. In the **Concurrent Prolog** compiler, the authors would like to implement a **Concurrent Prolog** goal's attempt to find a clause it can commit with by having the corresponding **Prolog** goal attempt to find a clause such that the single cut in its body may be successfully crossed. However, given a **Concurrent Prolog** clause, a **Concurrent Prolog** goal may either succeed or fail or *suspend*; given a **Prolog** clause, a **Prolog** goal may either succeed or fail. Hence of necessity the failure of a **Prolog** goal must correspond to either failure or suspension of the **Concurrent Prolog** goal.[5]

We fix this problem in our system by ensuring that (groups of) normal clauses are called only when there is no possibility of suspension for the goal and clauses in that group. Hence if control were ever to reach the 'postlude' clause, it could only be because of a failure to unify.

2.4.1 Unsuitability for cp(↓, | ,&)

Handling of suspended goals is an important reason why this scheme is not suitable for the implementation of cp(↓, | ,&), without drastic modifications. In cp(↓, | ,&), it may happen that some clauses for a predicate are '&'-clauses and some are ' | '-clauses. When a goal is invoked, it is possible that one of the '&'-clauses is able to commit. Later, execution downstream may fail because of some other goal failing. When computation 'backtracks' to this goal, alternate clauses must be examined. However, it is possible that the goal *suspends* on some of those clauses, and hence needs to be requeued. When it is next scheduled for execution, care needs to be taken to ensure that the '&'-clause which has already been tried is not tried again; otherwise the semantics of the language would be violated.

In our implementation, this information is maintained (sometimes implicitly) as a *modelist*, which specifies which modes (collections of clauses with identical suspension conditions) are still available for the goal to reduce against.

2.4.2 Scheduling

The Ueda implementation provides 100-*bounded depth-first* scheduling as the default strategy for the user.

N-bounded depth first scheduling means that each goal taken from the front of the queue is associated with a bound N. If it can unify aganst the head of a clause and successfully reduce the guard, then the goals in the body are all $N-1$ reducible, and are pushed in front of the goal queue. If a goal is 0-reducible it is taken from the front of the queue and reenqueued at the end.

N-bounded scheduling suffers from the obvious problem in a language with guards, namely, what bound should the guards be given? A particular guard execution may make more reductions than a constant bound, to terminate successfully. In their scheme, a call to a predicate with such a guard system

[4]pun not intended!

[5]There is one way out, which I once contemplated for **cp(↓, |)**, but gave up because it doesn't generalise to **cp(↓, |,&)** When control reaches the postlude clause, all that is needed is a single bit of information generated from the failed attempts to use the preceding clauses: namely whether any of those failures were due to suspension conditions not holding. The problem is that **Prolog** wipes out on backtracking, as it must, all the state it had generated on the failed branch. So I looked for some way in which a single bit of information can be stored in a **Prolog** system, across backtracks. (Of course, **retract** and **assert** are too costly.) It turns out that (in Dec-20 **Prolog**, and in most **Prolog** systems fashioned after it) **prompt/2** can be used precisely for this purpose, as can **op/3** and **current_op/3**.

could erroneously *fail*,[6] even though it should succeed, and *would* perhaps succeed with a higher bound. Consequently, this scheduling policy is useable only with *flat* guards, in which only calls to built-in predicates are allowed in guards. (Such calls are presumed to take unit time to execute.)

This is not a problem in our design because we adopt depth-first scheduling. This allows wait-free guards of arbitrary (unbounded) run-time complexity.

2.4.3 Modes

The prelude is optional, and is generated from extra 'mode information' which may be supplied by the user.

The user is allowed to declare one of three possible 'modes' for each top-level argument of a **Concurrent Prolog** predicate. If the mode is '+', the system generates code in two levels, so that any structure-level indexing provided by the underlying **Prolog** implementation may be exploited. If the mode is '?', no special processing is done. If the mode is '−' ('output'), the system takes that to be a declaration that the goal argument in that position will always be an 'uninstantiated non-read-only variable'. The authors point out that addition of mode information always seems to help: declaration of output mode seems particularly beneficial. In this case, calls to the general read-only-unification mechanism are replaced by the usual **Prolog** head unification.

In our design, there is no notion of user-supplied mode information for cp programs, for two reasons: it is not needed, and we have a different view of how such user-supplied declaration should be used. First, our implementation transparently (and always) ensures that user programs take advantage of whatever indexing scheme the underlying **Prolog** system provides. No declarations are necessary. Second, we feel that such user supplied 'pragmas' (see also declarations in CommonLisp) should only be used to enhance the performance of the implementation: in particular, their usage should not threaten the integrity of the implementation. The implementation of 'output mode' in the **Concurrent Prolog** compiler is incorrect in case the argument does turn out to have a read-only variable.[7] This can be a serious problem in practice because of the dynamic nature of the read-only variable. The amount of (global) analysis the user would have to do to guarantee that the output mode will not be violated could be quite prohibitive for reasonable (as opposed to these toy benchmark) programs. (All usages of that predicate would have to be examined, and for each usage an analysis of the possible bindings that can be generated at run-time would need to be done.) In fact, it is impossible to provide such a guarantee if that argument could ever be bound to a term supplied by the user at run-time in a query.

2.5 Other work

The related problem of compiling a non-flat language into a flat committed choice logic programming language has been examined in [Codish, 1985]. The notion of safety introduced there (see also [Takeuchi and Furukawa, 1986] or [Gregory, 1987], for example) has nothing to do with the notion of wait-free guards introduced in this paper. Safety as a concept seems important only to those languages

[6]In reality, it would suspend after it reaches its bound, and not set the deadlock flag. When all other goals have successfully executed, and this is the only goal left, the system would (eventually) schedule this goal, find that the deadlock flag is not set, declare deadlock and fail.

[7]It is incorrect for the following reasons: a query may deadlock when it should have succeeded, and a query may succeed when it should have deadlocked. It may be argued that our implementation of guards suffers from the same problem. However, if a guard turns out, in reality not to be wait-free, our implementation would only err on the safe side: a query might deadlock where it should have succeeded.

which do not wish to allow guard computations to bind variables in the call, or to those implementations concerned with compiling a language with non-flat guards into a flat language. Neither of these considerations hold in this context. Codish's work, however, does contain ideas relevant to the current paper, such as evaluating alternate clauses for a goal in an And-parallel environment, and using mutual exclusion to arbitrate between them.

The implementation of **Parlog** has been discussed in [Gregory, 1984], [Clark and Gregory, 1985] and most recently in [Gregory, 1987]. Like us, they present their implementation in a series of steps, translating down to more primitive languages. Unlike us, they consider translations down to languages more primitive than even **Prolog**. They do not have a notion of don't know commitment in the language. (They have a separate set-constructor interface to interact with a **Prolog** style 'all solutions' mechanism.) I am not aware of any published results on the performance of their implementation schemes.

[Miyazaki *et al.*, 1985] discusses an alternate sequential implementation of **Concurrent Prolog**, in which the focus is on explicitly representing and manipulating the Or-parallel environments necessary for an implementation of arbitrarily recursive guards in a CLP language with atomic commitment. We do not face this problem, because of the restriction to wait-free guards, as discussed later.

Recently, and independently, [Holmgren and Wærn, 1987] propose a somewhat similar scheme for implementing **GHC** in a **Prolog** with freeze. (It really works only for **FGHC**.) The notion of wait-free guards is more appropriate in the context of the cp languages. In addition, we present a notion of modes, and introduce single- and multi-mode optimisations. (They create a 'mode goal' for every clause in the definition of the predicate, resulting in avoidable overhead.) Our simple notion of a single exclusion chain provides a cleaner mechanism for ensuring that at least one mode goal commits. We also identify certain problems with current implementations of freeze in **Prolog** implementations. Since they are concerned with implementing **GHC**, they do not consider don't know commitment. We have also been concerned with arguing for the correctness of our scheme, and presenting it using a series of translations to more primitive languages. Also, they do not consider the related problem of compiling into a **Prolog** without a `freeze` primitive. Most **Prolog** implementations currently available do not have a built in implementation for `freeze`.

3 Implementing cp(↓,|, &) by translating into Prolog with freeze

We present the implementation in two steps. First we show how cp(↓, | ,&) programs may be implemented by translating them into cp(f, | ,&,o) programs. In overview, all the clauses of a cp(↓, | ,&) predicate are examined to determine the modes of the predicate. Informally, a mode is a collection of clauses which impose identical 'instantiatedness' constraints on a goal. In order to solve a cp(↓, | ,&) goal, a number of cp(f, | ,&,o) goals are generated, one for each mode. Each cp(f, | ,&,o) goal is delayed (using `freeze`) until such time as it satisfies the instantiatedness constraints of its mode. Then the goal is reduced using a set of cp(| ,&) clauses for the mode (obtained by forgetting the annotations in the corresponding cp(↓, | ,&) clauses), and an `otherwise` clause.

Next we show that cp(f, | ,&,o) programs have a remarkably direct implementation in **Prolog** with `freeze`, provided that the 'wait-free' restriction is placed on guards. Don't know commitment is implemented by exploiting **Prolog**'s inherent backtracking ability: because we do not specify Or-fairness, **Prolog**'s sequential search of alternative clauses with chronlogical backtracking is a sound implementation. Don't care commitment is implemented by using a cut. `otherwise` is implemented using **Prolog**'s clause sequential search rule.

In the remainder of this section, we introduce the concept of modes, discuss the code that is generated to implement the blocking condition of a mode, present the translation scheme and argue for its correctness, and present some enhancements to `freeze` functionality (in **Prolog** systems) motivated by the translation.

3.1 Modes

The starting point for the design is to identify the different *modes* for a predicate. A mode is a collection of clauses for a given predicate each of which have the same *blocking condition.*

Example 3.1 (Mode groups for `rev/2`**)** *Consider the following* cp(↓, | ,&) *program:*

```
rev([]↓, []).
rev([A|Rest]↓, Rev) ←  rev(Rest, ARev), append(ARev, [A], Rev).
```

Both the clauses have the same mode: the blocking condition for the mode is $\langle\{\epsilon, 1.\epsilon\}, \emptyset\rangle$. *Note that the blocking condition does not contain any information about the* terms *in the head of a clause: only the pattern of annotations.*

Indeed, most programs written in a determinate data-flow style have a single mode. `merge/3` *is a canonical example of a multi-mode predicate.*

Next we consider how to generate code to check the blocking condition of a mode. The definition of an annotated head allows mode-information to be generated top-down from a clause-head, by starting from the root, and at each level examining only the immediate subterms that (perhaps eventually) contain annotated subterms.

In the compiler, a blocking condition, generated by examining a clause head, is represented as a `mode/2` structure. The predicates that generate the structure from a clause head are straightforward and are omitted.

Definition 3.1 (The mode/2 data-structure) *A valid* `mode/2` *structure is a finite tree, satisfying the constraints imposed by the following grammar rules:*

```
Mode  ::== [Inst*]|mode([Inst*],[Equa+])
Inst  ::== inst(Num,Arit)|arg(Num,Var,Arit)
Arit  ::== arity(Num,[Inst+])|[]
Equa  ::== [Var+]
```

Here the two terminals are Num, which stands for a natural number, and Var, which stands for a variable. As usual for BNF formalisms, '⋆' denotes zero or more repetitions, and '+', one or more. The data-structure `inst(Num, Arit)` *represents the information that the* `Num`*th son of the current node must be instantiated. If* `Arit` *is* `[]`, *then the instantiated son does not contain any annotated subterms; otherwise,* `Arit` *is of the form* `arity(Num, Subtree)`, *where* `Num` *is the maximum of the set of indices of annotated children of that node, and* `Subtree` *represents the (recursive) annotation specifications for that node. The data-structure* `arg(Num, Var, Arit)` *represents the information that the* `Num`*th son of the current node is to be equated (at run-time) to* `Var`. `Arit` *is as above.*

The suspension code generated for a blocking condition is best defined by structural induction over the blocking condition. However, that is precisely how the *actual* program is structured. We feel that the definition of the code corresponding to the blocking condition is sufficiently close to its *implementation* as a definite clause program for us to present the program directly.

Definition 3.2 (Suspension Code) *In the following, we define a predicate* `code/3` *such that* `code(A, B, C)` *holds iff for a (non-variable) goal* `B`, *blocking conditon* `A` *may be implemented by executing* cp(f,|,&,o) *code* `C` *at runtime. The* cp(f,|,&,o) *'builtin' predicates needed to construct the code are:* `arg/3`, `functor/3` *and* `>=/2`. *(Their definitions are the same as in* **Dec-20 Prolog**.*) In order to construct a* non-blocking *version of the code that checks the instantiatedness conditions (i.e. this code simply succeeds if the blocking condition is satisfied, and* fails *otherwise), calls to* `freeze/2` *are transformed into calls to* `nonvar/1`, *and calls to* `freeze/3` *are transformed into calls to* `==/2`.

The following rules cover the three different types of modes:

```
code(A, B, C) ←  ccode(A, B, C\true).
ccode([], Head, Code \Code).
ccode([X|Rest], Head, Code\Code_1) ←
     icode(Head, [X|Rest], Code\Code_1).
 ccode(mode(Inst, Env), Head, Code\Code_2) ←
     icode(Inst, Head, Code\Code_1),
     ecode(Env, Code1\Code_2).
```

The predicate `ecode(A,B)` *holds iff* `A` *is a list of lists and* `B` *a goal, represented as a difference structure. Each list in* `A` *must be a list of variables. The goal is obtained by nesting the code produced from each each list in* `A`. *The code produced from a list* `[T_1,...,T_n]` *is* `freeze(T_1, T_2, freeze(T_2, T_3, ..., freeze(T_(n-1), T_n, L)...))`. *where* `L` *is the tail of the difference structure. Note that* $n \geq 0$.

```
ecode([], Code\Code).
ecode([L|R], Code\Tail) ←
     equate(L, Code\Code_1), ecode(R, Code_1\Tail).
 equate([L], Code\Code).
equate([A,B|R], freeze(A, B, Code)\Tail) ←
     equate([B|R], Code\Tail).
```

We distinguish between icode/3 *called at the top-level and called recursively within itself. The recursive calls are to* iacode/3. *When it is called at the top-lvel, we know that the goal itself (the call's second argument) is instantiated. Hence code need not be generated to obtain the Kth arg of the head – rather we can execute this call right now, at compile time.*

```
icode([], Head, Tail\Tail).
icode([inst(Arg,Subtree)|Rest], Head,
      freeze(S, Code)\Tail
     )←
      arg(Arg, Head, S),
      iacode(Subtree S, Code\Code1),
      icode(Rest, Head, Code1\Tail).
 icode([arg(N,A,Subtree)|Rest], Head, Code\Tail) ←
     arg(N, Head, A),
     iacode(Subtree, A, Code\Tail1),
     icode(Rest, Head, Tail1\Tail).
```

The definition of iacode/3 *is just like* icode/3, *except that all* arg/3 *tests on the first argument must be made at run-time.*

```
iacode([], S, Tail\Tail).
iacode([inst(Arg,Subtree)|Rest], S,
       (arg(Arg, S, S1), freeze(S1, Code))\Tail
      )←
      iacode(Subtree, S1, Code\Code1),
      iacode(Rest, S, Code1\Tail).
 iacode(arity(N, Subtree), Head,
       (functor(Head, F, M), M ≥ N, Code)\Tail
      )←
      iacode( Subtree, Head, Code\Tail).
 iacode(arg(N, A, Subtree), Head,
       (arg(N, Head, A), Code)\Tail
      )←
      iacode(Head, Subtree, Code\Tail).
```

3.2 Translation scheme

Given the previous section, we will now assume that the r clauses for a cp(↓, | ,&) predicate have been partitioned into k mode groups. These clauses will be translated into $(r+k+1)$ cp(f, | ,&,o) clauses: 1 of **Type I**, r of **Type II** and k of **Type III**.

Type I cause. First, we introduce a clause for p/n. This will be the only clause for p/n in the cp(f, | ,&,o) program:

```
p(X1,...,Xn) ←
```

```
freeze([freeze(X,true),C1[p_1(X1,...,Xn,X,off\Y1)]]),
freeze([freeze(X,true),C2[p_2(X1,...,Xn,X, Y1\Y2)]]),
...,
freeze([freeze(X,true),Ck[p_k(X1,...,Xn,X, Y(k-1)\on)]]).
```

Here `Ci` is the freeze code corresponding to the blocking condition for mode `i`: by the notation `Ci[G]` we mean the term `Ti`, where `Ti\G` is the code difference structure generated from the blocking condition for mode `i`.

The variables `X` and `Yi` will be used for 'arbitration' as discussed below. `X` is used for mutual exclusion, and the `Yi` for 'single exclusion'. The predicates `p_i` have arity $(n+2)$ and are 'new' (they are assumed not to occur in the original cp(↓, | ,&) program).

The clauses for `p_i/(n+2)` are obtained as follows. The clauses consist of m_i cp(| ,&) clauses, where m_i is the number of clauses in the input program of (with?) mode i, and one `otherwise` clause.

Type II clauses. The m_i cp(| ,&) clauses for `p_i/(n+2)` are obtained from the m_i cp(↓, | ,&) clauses for mode-group `i` as follows: to each clause of the form

```
p(A1,...,An)  ←  RHS.
```

corresponds a clause

```
p_i(A1',...,An',i,_)  ←  RHS.
```

in the translated program, where `Aj'` is the argument `Aj`, with all annotations removed, for $1 \leq j \leq k$. Note that `i` is known at compile time: the actual code that is generated will have a small integer constant in place of the 'meta-variable' `i`. (In what follows, each of the r such clauses will also be called a 'normal' clause.)

Type III clause. The single `otherwise` clause for each mode is of the following form:

```
p_i(X1,...,Xn,_,Y\Y)  ←  otherwise | true.
```

3.3 Discussion of the translation

Here is how the translation works. The first clause (for `p/n`) invokes the freeze goals. The freeze goals achieve 'Or-parallelism': they cause suspension until such time as the appropriate arguments for the mode group are instantiated. When an `p_i` goal is unfrozen, all the clauses for that mode are tried. One of the following cases must happen:

Case 1. Suppose some other sibling `p_j` goal ($j \neq i$) has reduced via a normal clause. In this case `X` is equated to `j`, and the only selectable clause for the `p_i` goal is the last one.

Note that all the goals p_o (for $o \neq j, 1 \leq o \leq k$), which have been activated because their blocking condition has now been met, will be able to reduce via their last clause, and only via their last clause. This corresponds to allowing only one Or-sibling guard system to commit in any one branch of the computation. Note also that once some p_j goal reduces via a normal clause, all the `freeze/1` goals corresponding to the other modes can be activated in that branch (resulting from commitment) in which the results of commitment are published. For these goals the freezing condition for the second alternative is met, even if the blocking condition for the mode is not yet met. Hence these goals will successfully terminate.

Case 2. Suppose no other sibling p_j goal ($j \neq i$) has reduced via a normal clause. In this case, the goal may commit to one of its normal clauses. This commitment mirrors the commitment of the corresponding goal to the corresponding clause in the original cp(↓, | ,&) program. One of the following must happen:

Case 2.1. Suppose the goal commits to a ' | ' clause. If a failure is subsequently encountered this branch of the computation will fail.

Case 2.2 Suppose the goal commits to a '&' clause. In this case, the current resolvent is split into two. As discussed above, because of mutual- and single-exclusion, the other mode goals for the predicate cannot commit in the branch in which the results of the commitment are published. In the other branch, note that the alternate choice points in guard computation, if any, are preserved, exactly as needed, together with the alternate choice points (clauses) for the committing goal if any, together with the other mode goals, that have not yet terminated, if any. The alternate clauses for the p_i goal will be tried in this alternate world at some future point, depending upon the scheduling of Or-parallel computations.

Finally, suppose p_i is not able to commit with any normal clause. Then the *otherwise* clause becomes applicable.[8] This clause will succeed iff less than $(k-1)$ of the other sibling p_j goals have succeeded with their last clause. (Only in this case will the Y\Y equating succeed; otherwise it will fail on attempting to equate on with off.) This ensures that the original p/n goal is able to reduce successfully in any Or-parallel world iff exactly one of the p_o goals reduces via a normal clause.

3.3.1 Optimisation for single-mode predicates

Note that in case a predicate has only a single mode group, the overhead of distributed commitment may be avoided. The single Type III clause will never be applicable, and can hence be omitted. But then, so can the two extra arguments added for single and mutual exclusion. Consequently, for single mode predicates, r cp(↓, | ,&) clauses for a predicate p/n translate into $(r+1)$ cp(f, | ,&,o) clauses, one for predicate p/n and r for predicate p_1/n.

This optimisation is important because in reality a large fraction of the code that is written seems to involve singel mode predicates. Certainly, all deterministic programs writen in the data-flow style, as transducers of streams of values (Kahn-style networks) will consist of single-mode fcp(↓, |) programs.

If the input program is further a cp(| ,&) program as well, then the blocking condition of the single mode group will be $\langle \emptyset, \emptyset \rangle$. Now the **Type I** clause may also be omitted. Only the r **Type II** clauses

[8]Note that head-unification cannot suspend for any clause for p_i, and neither can guard computation, because of the *wait-free* assumption. Hence all normal clauses will become non-candidates without having to wait for more information from the environment.

corresponding to the *r* cp(|,&) clauses are needed. This means that cp(|,&) programs are translated into identical cp(f,|,&,o) programs!

3.3.2 Example of translated code

Sample cp(↓,|,&) code and its translation into cp(f,|,&,o).

Consider a single mode cp(↓,|) predicate. The code and its translation is given below.

```
rev([X|Xs]↓, Ys) ←            rev(X, Y) ←
     rev(Xs, Zs),                  freeze(X, rev_1(X, Y)).
     append(Zs, [X], Ys).
 rev([]↓, []).                rev_1([X|Xs], Ys) ←
                                   rev(Xs, Zs),
                                   append(Zs, [X], Ys).
                               rev_1([], []).
```

The translation of a cp(|,&) predicate is the identity translation, so an example should not be necessary. The translation of a multi-mode predicate is as follows:

```
                              merge(A,B,C) ←
                                   freeze ([freeze(Flag, true) ,
                              freeze(A, merge_1(A,B,C, Flag, off\Y))]),
                              freeze ([freeze(Flag, true) ,
merge([A|X]↓, Y, [A|Z]) ←     freeze(B, merge_2(A,B,C, Flag, Y\Y1))]),
     merge(X,Y,Z).            freeze ([freeze(Flag, true) ,
 merge(X, [A|Y]↓, [A|Z]) ←    freeze(A,freeze(B,merge_3(A,B,C,
     merge(X,Y,Z).                                  Flag, Y1\on)))]).
 merge([]↓,[]↓,[]).            merge_1([A|X], Y, [A|Z], 1, _) ←
                                   merge(X, Y, Z).
                               merge_2(X, [A|Y], [A|Z], 2, _) ←
                                   merge(X, Y, Z).
                               merge_3([], [], [], 3, _).
                              merge_2(A, B, C, _, Y\Y) ←  otherwise | true.
                              merge_1(A, B, C, _, Y\Y) ←  otherwise | true.
                              merge_3(A, B, C, _, Y\Y) ←  otherwise | true.
```

3.4 Translating cp(f,|,&,o) programs into Prolog-with-freeze

We now consider the implementation of cp(f,|,&,o) programs by translation into **Prolog**.

We proceed in a series of steps. First, it should be clear that any cp(&) program may be implemented as a pure **Prolog** program as follows:

cp(&) *program:*	**Prolog** *program*
`Head ← Guard & Body.`	`Head ← Guard, Body.`

Since there is no possibility of suspension, a left-to-right, depth-first search strategy (as implemented in **Prolog**) is an adequate implementation.

It should also be clear that any cp(| ,&) program may be implemented in pure **Prolog**-with-cut, by translating ' | '-clauses as follows:

cp(|) *program:*	**Prolog** *program*
`Head ← Guard	Body.`	`Head ← Guard, !, Body.`

Next, consider cp(| ,&,o) programs. Suppose the clauses in the cp(| ,&,o) program have been rearranged so that all `otherwise` clauses textually succeed all other clauses for the same predicate. Such a cp(| ,&,o) program may be implemented in **Prolog** by using the two translation rules above, together with the following clause defining the predicate `otherwise`.

```
otherwise.
```

Finally, consider cp(f, | ,&,o) programs. Unfortunately, in general, the obvious translation (i.e. translate cp(f, | ,&,o) clauses according to the translation for cp(| ,&,o), and use the implementation of `freeze` in the host language) does not work, essentially because of the *guard-isolation* and *deep-wait* problems.

Recall that a cp clause can commit to a goal only when the guard has been *fully solved*. This means that before the **Prolog** clause can 'commit', it must ensure that no frozen goals generated during guard execution are left to be executed. Also, it is incorrect to execute any siblings of the parent goal that might be awakened because of bindings generated during guard execution, until guard execution has terminated successfully. This is the guard-isolation problem.

This problem can be handled by two assumptions. First, we assume the implementation supplies a primitive predicate, say `prolog/1` such that a goal `prolog(X)` suceeds iff its argument is a sequence of goals, and execution of the goals succeeds, with no frozen goals remaining. The sequence of goals `G` in the guard of a cp(f, | ,&,o) clause may then be executed as the goal `prolog(G)` in the 'guard' of the corresponding **Prolog** clause. Second, we must make some assumptions about when 'reawakened' goals are scheduled for execution.[9] We assume that reawakened goals are scheduled 'as late as possible', i.e. only after execution has terminated (perhaps with some frozen goals) at the current 'level', where each nested invocation of a `prolog/1` goal counts as a 'level'. The reawakened goals may be scheduled in an arbitrary order.

The second, more important, problem has to do with the case in which execution of the goals in the guard terminates, there are no reawakened goals, and some goals remain frozen. In cp, such a case corresponds to one branch of the guard system being blocked. If the guard computation contains a goal which &-committed to a clause, then there could be more than one disjunctive guard-system generated from the same guard. It is possible that one of those guard-systems has a successful execution sequence, and hence the **Prolog** implementation must make sure that it examines all the disjunctive guard systems until it finds one that is successful. Since we implement don't know commit by backtracking, it becomes necessary for the **Prolog** system to treat this case as *failure* and initiate backtracking.

During this backtracking the implementation may discover a successful execution of the guard system and commit. However, if the clause (containing the guard) happened to be a '&'-clause, then computation may later backtrack into the guard because of a deep failure encountered on scheduling some other (sibling of the parent) goal. If no other successful guard executions for the clause remain, it now becomes necessary to suspend the parent goal, since it is possible that its suspended guard-system branch could be reawakened

[9]Reawakened goals are **freeze** goals whose suspension condition has just been satisfied.

if some sibling of the parent goal is scheduled next. The parent goal must be suspended in such a way that information is recorded that prevents it from committing to those guard system invocations (and within an invocation, the specific branches!) to which it has already committed, if the semantics of &-splitting is to be preserved. But suspending such a partially executed goal is exceedingly complicated, for a variety of reasons. State (the frozen guard-system branch) would need to be stored across backtracks in the **Prolog** system. The size of the state could be arbitrarily large (guard execution could have suspended at an abitrary depth). This problem seems unsolvable in this framework.[10]

For the purposes of this paper, we will deal with the *deep-wait* problem by restricting the input language of the translator to **Prolog**-with-freeze to be cp(f, | ,&,o) with *wait-free* guards (Section 2.1). Such programs come with the assurance that the execution of no guard system can block on variables global to the parent goal. Hence, when executing a guard system, if a branch terminates with no reawakened goals but some frozen goals, it is safe to treat this as *deadlock* (i.e. a situation in which no progress is possible because each goal is blocked on others, with no possibility of any goal being resumed), fail and initiate backtracking to search for an alternative guard system branch, or alternative guard invocation.

This restriction implies that the cp(↓, | ,&) compiler produces code that is guaranteed to work correctly only for cp(↓, | ,&) programs with wait-free guards. In essence, the blocking condition for the mode defines the blocking condition for head unification and for *guard execution* for all the clauses in that mode. If a cp(↓, | ,&) program that does not satisfy this condition is input to the compiler, then the resulting progam is a potentially *incomplete* implementation. It is possible that some valid execution sequence for the cp(↓, | ,&) program is not discovered by the implementation.

To sum up. cp(f, | ,&,o) programs, with wait-free guards, may be implemented in a **Prolog**-with-freeze by virtue of the following translation:

- Sort all the cp(f, | ,&,o) clauses so that, for every predicate, all the `otherwise` clauses (if any) appear after the other clauses.
- Translate every cp(f, | ,&,o) clause into a **Prolog** clause according to the following table:

cp(f, | ,&,o) *program:*	**Prolog** *program*
`Head ← Guard & Body.`	`Head ← prolog(Guard), Body.`
`Head ← Guard | Body.`	`Head ← prolog(Guard),!, Body.`

The predicate `otherwise` is trivially implemented as discussed above. The definition of `prolog/1` is now summarised. A goal `prolog(X)` suceeds iff its argument is a sequence of goals, and execution of the goals succeeds, with no frozen goals remaining. If execution of goals terminates, and some goals are left frozen, then the execution is to be regarded as terminating in failure, and backtracking is initiated. On being backtracked into, the goal `prolog(X)` succeeds as many times as there are alternative solutions for X.

3.4.1 Performance Considerations

Note that according to the translation schemes given above, cp(| ,&) programs are translated to identical cp(f, | ,&,o) programs, which are converted into **Prolog** clauses that are nearly identical to the original

[10]A solution may, however, be possible for cp(↓,|,o). In this language, since any goal can have at most one successful execution, it would be sufficient to simply suspend the parent goal, and retry *all* the guard-systems for the goal again, at some future instant, as in the Ueda and Chikayama scheme. This scheme would certainly be wasteful in run-time because an arbitrarily large guard may need to be scheduled again and again. I have not explored this in detail.

cp(| ,&) clauses. The implementation for such programs pays no overhead for checking suspension conditions (because there are none), and runs essentially at the speed of the underlying **Prolog** implementation. This is significant because there *are* many cp(| ,&) programs. If the computational model of a cp progams is that of a network of cells performing local computation and communicating with other cells, then the predicates which spawn the network to begin with are often cp(| ,&) programs.

It is not possible to take advantage of such syntactic properties of subsets of the language for languages with 'dynamic annotations' such as **Concurrent Prolog**. Even if no clause for a predicate has any occurrences of '?', at run-time a read-only annotated variable may still be passed in. Hence it is impossible to implement the program directly as a **Prolog** program, without including the general (and somewhat cumbersome) machinery for read-only unification.

Another important advantage of the above translation scheme is that good use maybe made of the underlying **Prolog**'s indexing and/or determinacy detection and/or mode declaration facilties. In most **Prolog** systems, the implementation of head unification is optimised: all these optimisations are directly applicable for cp because cp unification (without the annotations) gets implemented as **Prolog** head unification. If the underlying **Prolog** system accepts it, nested mode declarations may also be generated by the compiler. For example, from the clause

```
merge([A↓|X], B, [A|Y]) ←  merge(X, B, Y).
```

the mode declaration

```
:-  mode merge_1(+(+,?), ?, ?).
```

may automatically be generated. (As an aside, I have observed that it is not uncommon to write cp(↓, | ,&) programs in which terms are nested to a depth of 3 – 4 in the head of a clause.)

3.5 On requirements from a freeze implementation

I now discuss and summarize some requirement on the functionality of freeze (in **Prolog** systems) that should have been apparent from the discussion above.

Activation of awakened freeze goals One of the basic problems with the current implementation of freeze is the '*cut capture*' problem. If a goal g is frozen on a variable X, and if X is bound during resolution (head unification) of goal g1 with clause c1, then g will be invoked immediately after head unification. If c1 has a cut in its body, then this cut will also cut away alternate solutions to the frozen goal. This is unacceptable from our viewpoint: the solutions to the frozen goal have nothing to do with the goals in the body of c1, and hence the proof of g1.

Finally, as discussed above, it should be possible to allow for goals to be frozen pending two terms becoming equated, not just pending a variable becoming instantiated. (freeze/3 in cp(f, | ,&).) Similarly, it would seem useful to have the equivalent of freeze/1 in **Prolog**.

3.6 Performance figures

Because of some of the above problems with freeze, we have been unable to run cp(↓, | ,&) programs on top of a **Prolog**-with-freeze. However, the cut capture problem does not affect the implementation of fcp(↓, |) programs. Performance numbers for some such programs, running on **Sicstus Prolog**, Version

1.1 on a Vax-780 are presented below. These numbers are compared with the numbers for the **Flat Concurrent Prolog** implementation (**Logix**), and **Quintus Prolog** running on a Vax-750, and for a plain **Sicstus Prolog** version of the program running on a Vax-780. The numbers for **Logix** and **Quintus** are from [Silverman, 1986]. (All times are in msec.)

Test	*Implementation/Machine*			
	Logix/750	Quintus/750	FCP/780	SICS/780
nrev (100 elems)	2850	717	3350	790
quicksort (100 elems)	7570	5150/3	4040	1380
hanoi	5530	2716	1440	1440

4 Compiling into plain Prolog

We now turn our attention to an implementation that compiles into pure **Prolog**.

As before, we proceed in a series of steps. We first present a meta-interpreter for cp(f, | ,&; o) with wait-free guards, and show how it may be rewritten into a deadlock-detecting cp(f, | ,&; o) interpreter, written in cp(| ,&; o), by introducing the notion of a suspension queue. (The interpreter itself has wait-free guards.) In essence a goal can be solved by passing it a suspension list on which any `freeze` subgoal places itself when it must suspend because its blocking condition is not met. Hence if a goal returns successfully, it may also return some suspended subgoals which need to be solved completely in order to obtain a solution. At the top-level, these goals are then called in turn, until there are no more goals left (success) or else a goal fails irrevocably or else no progress can be made (deadlock).

The translation from cp(f, | ,&; o) programs with wait-free guards into **Prolog**-with-freeze given in the previous section is applicable to the sublanguage cp(| ,&; o) as well, giving a translation from cp(| ,&; o) to **Prolog**. Altogether, we thus get a deadlock-detecting interpreter for cp(f, | ,&; o) written in **Prolog**. Combined with the translation from cp(↓, | ,&) to cp(f, | ,&,o) discussed in the previous section, this gives us an interpreter for cp(↓, | ,&) in **Prolog**. Given a particular cp(↓, | ,&) program, this interpreter may be partially evaluated to produce a **Prolog** program that may be compiled and run by the **Prolog** system. The cp(↓, | ,&) compiler in fact produces the **Prolog** code directly.

4.1 A Deadlock-detecting interpreter for cp(f, | ,&,o)

4.1.1 Input language

The *input language* for the interpreter is a collection of `clause/1` '&'-facts. The argument to each fact is a cp(f, | ,&; o) clause, with wait-free guards. Hence a `clause/1` assertion is of one of the following types:

```
clause((Head  ←  Guard | Body) :- true & true.
clause((Head  ←  Guard & Body) :- true & true.
clause((Head  ←  otherwise | Body):- true & true.
```

An 'empty' guard or body is represented by the goal `true`.

4.1.2 Meta-interpreter for cp(f, | ,&; o)

The meta-interpreter is straightforward. It is an adaptation of the interpreter in [Saraswat, forthcoming]. Note that every object level operation is 'absorbed' at the meta-level, i.e. implemented by exactly the same operation.[11]

```
cp(true).
cp((G1,G2)) ←  cp(G1), cp(G2).
cp((G1; G2)) ←  cp(G1); cp(G2).
cp(Goal) ←  builtin(Goal) | system(Goal).
cp(Goal) ←  otherwise | user(Goal).
user(Goal) ←
     clause((Goal ← Guard | Body)),
     Guard ≠ otherwise; cp(Guard) | cp(Body).
 user(Goal) ←
     clause((Goal ← Guard & Body ));
     cp(Guard) & cp(Body).
 user(Goal) ←
     otherwise | clause((Goal :- otherwise |Body)); cp(Body).

builtin(freeze(X)).
builtin(freeze(X, Y)).
builtin(freeze(X, Y, Z)).
other builtins ...

system(freeze(X)) ←  freeze(X).
system(freeze(X,Y)) ←  freeze(X,Y).
system(freeze(X,Y,Z)) ←  freeze(X,Y,Z).
other axioms ....
```

It is worth noting that the only control construct that has really been used in an essential way in the meta-interpreter (i.e. to serve a purpose other than interpreting itself) is And-sequentiality: it is used to ensure that all calls to `cp/1` from within the program have their argument instantiated to a goal(s).

[11]For simplicity of presentation, we further impose the syntactic restriction that if an **otherwise** goal appears in a guard, then it may be the only goal in the guard. This restriction is not computationally significant.

4.1.3 A cp(| ,&; o) interpreter for cp(f, | ,&,o)

We now turn to the task of providing an implementation for `freeze` in cp(| ,&; o). As in Section 2.3, it becomes necessary to introduce extra state in the interpreter that corresponds to a suspension queue and a deadlock flag. When a `freeze` goal cannot be reduced because its blocking condition does not yet hold, it is placed on the suspension queue. The *deadlock flag* is a variable that is initially unbound. Once all goals have been reduced, the suspension queue is examined. If it is not empty, then the goals in the queue are executed with a new deadlock flag. This flag is set if the current cycle causes any 'significant event' to occur. For example, the reduction of a process in the current cycle would count as a significant event. If at the end of the current cycle, when all the goals on the suspension queue have been examined once, and no significant event has occurred, then it is safe to assume that no significant event *will* occur, and deadlock may be declared, if the queue is not empty. Otherwise, the cycle repeats.

At the top-level, a `GoalSystem` is accepted.

```
cp(true).
cp(X)  ←  otherwise | cp(X, Susp, D); termination(Susp, D).
cp((G1, G2), Head\Tail, D) ←
     cp(G1, Head\Middle, D), cp(G2, Middle\Tail, D).
cp(Goal, Susp, D) ←  builtin(Goal) | system(Goal, Susp, D).
cp(Goal, Susp, D) ←  otherwise | user(Goal, Susp, D).

user(Goal, Susp, nd) ←
     clause((Goal ← Guard | Body)),
     Guard ≠ otherwise; cp(Guard) | cp(Body, Susp, _).
user(Goal, Susp, nd) ←
     clause((Goal ← Guard & Body));
     cp(Guard) & cp(Body, Susp, _).
user(Goal, Susp, nd) ←
     otherwise |
     clause((Goal ← otherwise |Body)); cp(Body, Susp, _).

builtin(true).
builtin(freeze(X)).
builtin(freeze(X, Y)).
builtin(freeze(X, Y, Z)).
other builtins ...

system(freeze(X), Susp, D) ←
     freeze(X, freeze(X), Susp, D).
system(freeze(X, Y), Susp, D) ←
     freeze(X, Y, Susp, D).
system(freeze(X, Y, Z), Susp, D) ←
     freeze(X, Y, Z, freeze(X,Y, Z), Susp, D).
```

We now consider the implementation of the builtin predicates.

Freeze A call `freeze(X)` is implemented by a call of the form `freeze(X, freeze(X), Susp, D)`.[12] If the blocking condition of any element in the list X is satisfied then the deadlock flag is set and the corresponding goal executed, with the same suspension queue. When none of the branches succeed, the goal is reenqueued on the suspension list. Note that only successful execution of a blocking condition qualifies as a significant event. This is the only goal that places itself on the suspension queue.

```
freeze([], Goal, (Goal, Tail)\Tail, D).
freeze([freeze(C,G)|R], Goal, Susp, nd) ← nonvar(C) | cp(G, Susp, _).
freeze([freeze(A,B,G)|R], Goal, Susp, nd) ← A == B | cp(G, Susp, _).
freeze([_|R], Goal, Susp, D) ← otherwise | freeze(R, Goal, Susp, D).
```

Calls to `freeze/2` and `freeze/3` are implemented similarly. We illustrate `freeze/2`:

```
freeze(C, Goal, Susp, nd) ← nonvar(C) | cp(Goal, Susp, _).
freeze(C, Goal, (freeze(C, Goal),Tail)\Tail, D) ← otherwise | true.
```

Checking termination When the marker is checked, it may result in the discovery of deadlock, termination or that there are more goals to be solved. Deadlock occurs if all the goals in the current suspension list have been exmined without any one reducing. Otherwise the remaining goals are executed.

```
termination(Susp\true, deadlock) ← display(deadlock(Susp)) | fail.
termination(Head\true, nd) ← otherwise | cp(Head).
```

Other builtins Other 'built-in' goals are executed by simply invoking the corresponding primitive goals in cp(| ,&; o), wherever necessary. (This includes the goal `true`.) For example:

```
system(true, Head\Head, nd).
system(A < B, Head\Head, nd) ← A < B.
```

Executing a builtin goal is always presumed to be a significant event. (Hence the third argument in the goal is equated to `nd`.)

4.1.4 Properties of the interpreter

This interpreter satisfies the following propositions:

- The cp query succeeds, and never loops iff (for this query) the implementation succeeds (with all the same answers), and never loops.
- The cp query always fails iff the implementation fails.
- The cp query deadlocks iff the implementation reports deadlock.

[12]A similar implementation is described in [Cohen, 1985].

4.2 Translating cp(| ,&; o) to Prolog

In Section 3.4, we showed how cp(| ,&,o) programs may be translated almost trivially into **Prolog**. The translation is considerably simpler for cp(| ,&; o), since there is now no possibility of suspension. A goal system of the form `G1 ; G2` may simply be translated into the (**Prolog**) goal system `G1, G2`. More precisely:

- Sort all the cp(| ,&; o) clauses so that, for every predicate, all the `otherwise` clauses (if any) appear after the other clauses.
- Translate every cp(| ,&; o) clause into a **Prolog** clause according to the following table:

cp(| ,&; o) *program:*	**Prolog** *program*
`Head ← Guard & Body.`	`Head ← T{Guard}, T{Body}.`
`Head ← Guard | Body.`	`Head ← T{Guard}, !, T{Body}.`

where

$$\mathtt{T\{G\}} = \begin{cases} \mathtt{T\{G1\},T\{G2\}} & \mathtt{if\ G=(G1,G2)\ or\ G=(G1;G2)} \\ \mathtt{G} & \mathtt{otherwise} \end{cases}$$

4.3 Code generated by compiler

We now exhibit the cp(| ,&; o) code generated by (an early version of) the compiler. We contend that this code may in principle be generated by partially evaluating the interpreter above, with an input cp(f, | ,&,o) program in the form of a `clause/1` predicate, and then applying some conceptually straightforward program transformation steps. However, even to present an informal derivation for a simple program would require the presentation of program transformation rules for cp(| ,&; o), which has to be beyond the scope of this paper. We hope to present such rules, and revisit this example in future work.

In what follows, we mix object- and meta-level notation in the following way. Object-level code will appear in `this` font, and meta-level code in *this* font. We use this notation, for example, when we want to talk of the different predicates, one for each mode, generated for a source predicate `p/n`: thus the syntax `p_`*i*`(A,B)` stands for an actual (object-level) call of the form `p_2(A,B)`, or `p_3(A,B)`, if the meta-variable *i* ranges over the set $\{2, 3\}$.

4.3.1 Single mode predicates

The following code is generated for a single mode predicate `p/n`:

Entry predicate. For a predicate `p/n`

```
p(A_1,..., A_n, H, T, nd) ←
     Code | p(A_1,..., A_n, H, T).
 p(A_1,..., A_n, p(A_1,..., A_n, R), R, D) ←  otherwise |true.
```

In the above `Code` is the non-blocking version of the code that checks the instantiatedness conditions for a mode. (Section 3.1.)

Real predicate. To every clause in the source program for `p/n` corresponds a clause in the object program for `p_1/(n+2)`.

First, some definitions. Given a (possibly annotated term) `T`, the *clean form* of `T`, notated by `T'`, is the term obtained by erasing all annotations in `T`. Given a conjunction

$$G \equiv p_1(A_1^1, \ldots, A_{n_1}^1), p_2(A_1^2, \ldots, A_{n_2}^2), \ldots, p_k(A_1^k, \ldots, A_{n_k}^k)$$

the `(H,T,D)` *chained form of* `G`, notated by `G[H,T,D]`, is the conjunction

$$\begin{aligned} G[H,T,D] \equiv\ & p_1(A_1^1, \ldots, A_{n_1}^1, H, M_1, D), \\ & p_2(A_1^2, \ldots, A_{n_2}^2, M_1, M_2, D), \ldots, \\ & p_k(A_1^k, \ldots, A_{n_k}^k, M_{(k-1)}, T, D) \end{aligned}$$

where $M_1, \ldots, M_{(k-1)}$ are presumed to be 'new' variables, i.e. they do not occur elsewhere in the (enclosing) clause.

Each clause in the source program is translated according to the following scheme:

cp(↓,∣,&) *program:*	cp(∣;&) *program:*

`p(A_1, ..., A_n) ←`
`Guard % Body.`
	`p_1(A_1',...,A_n', H, T) ←`
	`Guard[Gh,true,GD]; execute(Gh,GD)%`
	`Body[H,T,D].`

`p(A_1, ..., A_n) ←`
`Guard % Body.`
	`p_1(A_1',...,A_n', H, T) ←`
	`Guard% Body[H,T,D].`

The second form is a simpler version of the first, and can be chosen only when `Guard` consists solely of calls to built-in predicates, calls to which cannot suspend. Note that the variable `D` does not occur either in the head or the guard of the clause.

Single mode predicates, with vacuous mode. For such predicates the *Entry predicate* clause is omitted. For the *Real predicate* clauses, the name of the actual predicate (instead of a subscripted variant) is used. Further, note that the `Deadlock` flag does not need to be passed to such goals since no goal for such a predicate can ever be placed on the suspension queue. (Recall the `Deadlock` flag is needed to enable the detection of the situation in which all goals have been scheduled from the suspension queue, and none is able to make any significant progress towards reduction.)

Wait-free predicates. The general scheme above may be specialised to the following important subcase.

Given a cp(↓,∣,&) program, a *wait-free predicate* is a single mode predicate with vacuous mode such that for every clause for the predicate, every call in the body is, recursively, for a wait-free predicate.

Note that this class of programs is a superset of the set of cp(∣,&) programs, since a guard may contain calls to non-wait-free predicates. This is not a problem under the asusmption that the guard itself is wait-free.

Wait-freeness of a predicate is a property that may be computed very simply at compile time as follows. The *call dependency graph* for a program is a directed graph with nodes labelled by predicates, and an edge from node `p_i` to node `p_j` iff there exists a clause for `p_j` which contains a call to `p_i` in its body. Consider a call dependency graph in which a node is *coloured* iff it corresponds to a built-in predicate calls to which cannot suspend, or to a user-predicate which is not a single mode predicate with vacuous mode. A predicate is wait-free iff its node is uncoloured in the transitive closure of this graph.

By definition, a wait-free predicate has the property that, on any call, neither the call (nor any subgoal generated by the call) may suspend, regardless of the scheduling policy used to execute the call and its subgoals. Hence, according to our depth-first scheduling strategy, a successful call to such a predicate will always result in the `Head` and `Tail` variables (which represent the suspension queue) becoming equated. Hence for such predicates, in addition to the `Deadlock` argument being omitted (as above) the `Head` and `Tail` arguments may also be omitted. In particular, this means that for such a source-predicate `p/n`, the cp(|;&) predicate that implements it is also `p/n`. If a call is made to `p/n` in some clause in the source program, this call is translated into a call to `p/n` in the target program (i.e. it is not 'chained into' the `Goalsystem` it occurs in).

Finally, note that the second form of the translation given above for the real clauses of the cp(↓,|,&) program may be strengthened as follows. The cp(↓,|,&) program clause

```
p(A_1, ..., A_n)   ←   Guard % Body.
```

may be translated into the clause:

```
p_1(A_1',...,A_n', H, T)   ←   Guard %Body[H,T,D].
```

if every call to a predicate in `Guard` is either for a built-in predicate that does not suspend, or for a wait-free predicate.

Whenever the two conditions discussed in this paragraph are satisfied, the predicate may be implemented by simply translating its clauses into the corresponding **Prolog** clauses. Not even extra arguments are necessary. Extra arguments have been found, experimentally, to slow down execution by 30% in some cases.

4.3.2 Multi-mode predicates

Entry predicate. The entry predicate for a source predicate `p/n` is a predicate `p/(n+3)`. The three extra arguments are the `Head` and `Tail` of the susension queue and the `Deadlock` flag. There is only one clause for `p/(n+3)` (the **Type I** clause), whose body contains calls to *k freeze predicates*, where *k* is the number of modes for `p/n`. These calls are strung on two chains: a suspension chain terminating in `Head` and `Tail`, and a single exclusion chain terminating in `off` and `on`. In addition each call shares the `Deadlock` flag, and also a mutual exclusion `Flag`.

```
p(A_1,..., A_n, Head, Tail, Deadlock) ←
      p_f_1(A_1,...,A_n, Head, T_1, Deadlock, Flag, off, M_1),
      p_f_2(A_1,...,A_n, T_1, T_2, Deadlock, Flag, M_1, M_2),
      ...,
      p_f_k(A_1,...,A_n, T_(k-1), Tail, Deadlock, Flag, M_(k-1), on).
```

Freeze Predicates. For each mode *i*, the following clauses are defined for the freeze predicate `p_f_i/(n+6)`. The clauses can be thought of as being obtained by specialising a call to `freeze/1` when its single argument is known to be a list of two alternatives. We use explicit clause-sequencing in lieu of nested `otherwise` goals.

```
p_f_i(A_1,..., A_n, Head, Head, Deadlock, Flag, M, M) ←
     nonvar(Flag) | true;;
p_f_i(A_1,..., A_n, Head, Tail, Deadlock, Flag, M_1, M_2) ←
     Code_i | p_i(A_1,..., A_n, Head, Tail, Flag, M_1, M_2);;
p_m_i(A_1,..., A_n,
     p_f_i(A_1,..., A_n, Tail), Tail, Deadlock, Flag, M, M
    ).
```

Real predicate. A freeze predicate for mode *i* contains a call to the real predicate for mode *i*. The clauses for the real predicate are the **Type II** and `Type III` clauses. The **Type II** clauses are obtained by a one-to-one translation of the clauses in the source program in mode group *i*, for predicate `p/n`. The translation is analogous to the translation for real predicates for single-mode predicates, except that the extra argument in the head have to be handled correctly:

```
cp(↓,|,&) program:   cp(|;&) program
p(A_1, ..., A_n) ←   p_1(A_1',...,A_n', Head, Tail, i, _, _) ←
    Guard % Body.         Guard[GH, true, GD]; execute(GH, GD)%
                          Body[Head, Tail, D].
p(A_1, ..., A_n) ←   p_1(A_1',...,A_n', Head, Tail, i, _, _) ←
    Guard % Body.         Guard% Body[Head, Tail, D].
```

As before, the second form is a simpler version of the first, and can be chosen only when `Guard` consists solely of calls to either those built-in predicates for which calls cannot suspend, or to wait-free user-defined predicates. Note too that if the guard or body contains goals for wait-free predicates, then these goals are not 'chained in'. Also, goals for single mode predicates with vacuous modes (which are not also wait-free) are called only with two extra arguments (the `Deadlock` flag is dropped).

In addition, there is one `otherwise` clause for `p_i/(n+5)`, corresponding to the **Type III** clause:

```
p_i(A_1,..., A_n, Head, Head, Deadlock, Flag, M, M) ←
     otherwise  | true.
```

4.4 Optimisations for multi-mode predicates

When attempting a translation from cp(↓,|,&) to a **Prolog**-with-freeze, some kind of distributed commitment operations is essential. Using `freeze` results in non-busy waiting: a frozen goal is scheduled for execution, by a constant-time operation, after its blocking condition is met. It is not touched if its blocking condition is not met. However, the blocking conditions for different modes could be achieved at different instants, during the course of execution, and hence it is necessary to coordinate commitment of a goal to one of is clauses through shared variables. Moreover, it is necessary to spawn mode goals for *each* mode, since it is not known a priori which mode is going to contain a clause that will commit successfully.

Generating all the mode goals at once can be expensive in time and space, unavoidable as it seems for a `freeze`-based implementation. The amount of memory used may be calculated as follows: in the worst case, when the goal is first scheduled, none of the blocking conditions for its modes is satisfied. Hence k mode goals will be placed on the heap. Each mode goal will occupy at least $(n + 7)$ words (to store the goal for the real predicate) on a WAM-based **Prolog** implementation, discounting whatever space is consumed by the mechanism for implementing disjunctive freeze.[13] Moreover, once a clause commits, $O(k)$ time is necessary to unfreeze the other mode goals. This must happen on every clause commitment for the goal (multiple clause commitments are possible with '&'-clauses).

This overhead can be reduced when compiling into a (vanilla) **Prolog**. In this case, it is not possible to implement non-busy waiting simply. [14] Hence the blocking condition of a mode goal has to be checked blindly and periodically by the implementation. But advantage can be taken of this periodic checking by checking the blocking conditions of *all* the remaining modes for the goal at the same time. This leads to the idea of maintaining a *single* **Prolog** goal for every cp goal, and keeping within the goal, if necessary, a data structure (the *mode list*) representing all the available modes for the goal. (The available modes are those whose blocking condition has not yet been determined to hold.)

In overview the new scheme works as follows. To implement a multi-mode cp predicate, three kinds of **Prolog** predicates are needed: one *Enter Freeze* predicate, one *Real Freeze* predicate and k *Normal* predicates, one for each mode. A cp predicate is invoked by executing the corresponding Enter Freeze predicate. This predicate examines the blocking code for each mode sequentially, in alternate clauses. If the blocking code for no mode holds, then the predicate places itself on the suspension queue. If the blocking condition holds for any mode, the corresponding Normal predicate is invoked, with a data structure that represents the the *Past* and *Future* modelists. The *Past* list is the (conceptual) list of all those modes that have already been examined and for which the blocking condition does not yet hold. In this case, the Past list contains the list of all the modes upto but not including the current mode. The *Future* list is the (conceptual) list of available modes that have not yet been examined. In this case, the Future list contains the list of all the modes succeeding the current mode and upto the highest mode for the predicate.

The Normal predicate examines its **Type II** clauses in turn, searching for a clause that could be used to commit. If no clause can commit (or if some &-clauses can commit, but fail due to subsequent deep failure), the **Type III** clause for the Normal predicate is invoked. This clause invokes the Real Freeze predicate, with, among other arguments, the Past and Future modelist it was invoked with.

The Real Freeze predicate examines the blocking conditions for all the modes in its Future list in turn, using alternative clauses. If the blocking condition for a particular mode holds, it calls the corresponding Normal predicate, passing it its current Past and Future lists. If the blocking condition of a mode does not hold, the mode is stored in the Past list, and the next mode in the Future list is examined. If there is none, and if the Past list contains at least one mode, the Real Freeze predicate places itself on the suspension queue, with its Past list.

[13]The memory overhead could conceivably be brought down to $O(n)$ (from $O(k \times n)$) by *sharing* as much structure (e.g. the n arguments) across the mode goals as possible. However, if for indexing purposes, it is desirable to maintain the same order of arguments in the real predicate as in the source predicate **p/n**, then such sharing would involve a 'packing' and 'unpacking' phase, which could be expensive in time.

[14]One could imagine a non-busy-waiting implementation in **Prolog** in which **cp** variables are represented by **Prolog** structures that hold (for the unbound **cp** variables) the list of processes suspended on that variable. There are two major problems with this. One, **cp** unification can no longer be implemented by **Prolog** unification. This is likely to cause a major performance degradation. Second, **Prolog** structures, of course, are *not* mutable, only extensible. Hence when a new (**cp**) process needs to suspend on a **cp** variable, this process must be added by instantiating a pre-existing variable in the structure. This variable can at best be the leaf of a balanced binary tree, leading to non-constant cost just for suspending on a single variable. These two considerations are enough to make the proposal not competitive.

4.4.1 Allocating memory for mode lists

Mode lists essentially represent *choice points* information. In a language such as **Prolog**, which has only sequential search for clauses, with no possibility of suspension, the list of available clauses can be stored as a single pointer. Furthermore choice points are created on the stack, and can be deallocated inexpensively on backtracking.

In the current context, however, it is necessary to allocate mode lists on the heap. It now becomes quite important to avoid allocating more space than necessary because this space will usually become garbage quickly. With a stop and go heap garbage collector, this would mean that the machine would run out of space more quickly. With an *ephemeral* garbage collector, this would mean that computations would run more slowly as garbage is collected at a fixed fraction of the rate at which memory is consumed.

Considerable space savings can be made by representing the Past and Future lists *implicitly* wherever possible. We now discuss how to do this.

4.4.2 Interconversions between mode list types

A *mode* is represented as a constant drawn from the following set `m1, m2, m3,..., mk`, where `k` is an implementation dependent maximum limit.[15] The special mode `noalts` is used as a sentinel. The implementation maintains a database of `succ_mode/2` facts of the form:

```
succ_mode(noalts, m1).
succ_mode(m1, m2).
succ_mode(m2, m3).
...
succ_mode(mk, noalts).
```

The `succ_mode/2` predicate essentially represents a shareable modelist in code. A single mode `M` may be taken to stand for the (implicit) Past list of all the modes from `m1` upto (and not including) `M`. It may also be taken to stand for the (implicit) Future list of all the modes from `M1` to the highest mode for the predicate, where `succ_mode(M, M1)` holds.

This motivates representing modelists as a datastructure with three components: a mode list type, a Past component and a Future component. The mode list type is a constant in the set { `ii, ie, ei, ee`}. The first letter indicates the type of representation for the Past list, and the second the type of representation for the Future list. (`i` stands for implicit and `e` for explicit.)

When the mode list type is `ii`, the Future component contains a mode, which represents both the Past and Future lists implicitly, as discussed above. (The Past component may contain anything and is ignored.)

When the mode list type is `ie`, the Past component contains a list `P` of modes, and the Future component a list `Q` such that `Q` is a tail of `P`. In this case, the Past list is represented implicitly (as a kind of difference list): if `P` is of the form `[A_1,..., A_m, A_(m+1),A_(m+2),..., A_n]`, and `Q` is of the form `[A_(m+2),..., A_n]`, then the Past list is the list `[A_1,...,A_m]`. (`A_(m+1)` is the 'current mode'.)

When the mode list type is `ei`, the Past component is a list of modes (the Past list) and the Future component is a mode, `M`. The Future list is the Future list implicitly represented by `M`.

When the mode list type is `ee`, the Past and Future component both contain lists that are, respectively, the Past and Future mode lists.

[15]A mode is represented as a constant rather than as an integer for indexing purposes. See the syntax of Real Freeze clauses.

We now discuss when interconversions between these mode list types occur. The Enter Freeze goal invokes the Normal predicate with a mode list of type `ii`. If the Normal goal was invoked with mode list of type `ii`, it invokes the Real Freeze goal (when necessary) with a mode list of type `ei`, after constructing the explicit Past list. If the Normal goal was invoked with mode list of type `ie`, it invokes the Real Freeze goal (when necessary) with a mode list of type `ee`, after constructing the explicit Past list. The Real Freeze goal, when taken off the suspension queue, is invoked with mode list type `ie`. (A Real Freeze goal is always placed on the suspension queue with an explicitly constructed list of alternatives.)

In this way, all consing is avoided unless absolutely necessary: that is, unless the blocking condition for an available mode has been discovered to hold, and no clause in that mode group can commit irrevocably. In this case a new mode list has to be constructed which does not contain the mode that has been checked and discarded.

4.4.3 Performance analysis

In the following important cases no fresh mode list consing is done: the Enter Freeze (or Real Freeze) goal examines its (available) modes determines that the blocking condition for no mode holds, and suspends and is scheduled a number of times, before finally finding a mode whose blocking condition holds and committing irrevocably to a clause in that mode group. Furthermore, in the case of the Enter Freeze predicate above, *no* mode list consing is done at all.

In the worst case, each time an Enter or a Real Freeze goal is scheduled it loses one mode. In this case $O(k \times k)$ space will be used up for mode list consing.

4.4.4 Actual Code generated

We now exhibit the actual code generated for multi-mode predicates by the version of the compiler that optimises single- and multi-mode predicates. As before we assume that code is being generated for a source predicate `p/n` which has k modes and r clauses.

Entry predicate. The five extra arguments in a call to a Normal predicate, are, successively, the Head and Tail of the suspension list, the mode list type, the Past component and the Future component. We use explicit clause sequencing here, in lieu of a nested chain of `otherwise` clauses.

```
p(A_1, ..., A_n, H, T, nd) ←
     Code_1 | p_1(A_1, ..., A_n, H, T, ii, _, _);;
...
p(A_1, ..., A_n, H, T, nd) ←
     Code_k | p_k(A_1, ..., A_n, H, T, ii, _, _);;
 p(A_1,..., A_n, p(A_1, ... A_n, R), R, D).
```

Real Freeze Predicate. There are actually two predicates. When the Real Freeze predicate is invoked from the suspension queue, the predicate `p_f(n+4)` is called (for the source program predicate `p/n`). The four extra arguments are, in turn: the `Head` and `Tail` of the suspension queue, the `Deadlock` flag, and the list of alternative modes for the clause. This predicate internally calls the actual Real Freeze Mode predicate `p_f/(n+7)`. The seven arguments are, in sequence, the current mode `M`, the `Head` and `Tail` of the suspension queue, the `Deadlock` flag, and the three fields of the mode list representation. In reality, the current mode argument is the first argument to the call to take advantage of indexing on the first argument, provided by the underlying **Prolog**.

Header clause:

```
p_f(A_1,..., A_n, H, T, D, Alts) ←
     Alts =[M | RAlts]);
     p_f(M, A_1,..., A_n, H, T, D, ie, Alts, RAlts).

p_f(noalts, A_1,..., A_n, p_f(A_1,...,A_n,T, Fz), T,
    D, Flag, Fz, Future
   ).
```

And for each mode, Mi ∈ {m1,m2,...,mk}, *where* mk *is the highest mode for* p/n*:*

```
p_f(Mi, A_1,..., A_n, H,T, nd, Flag, Past, Future) ←
     Code_i | p_i(A_1,..., A_n, H,T, Flag, Past, Future).
...,
```

The following clause checks alternate modes.
MK+1 *is the (compile-time known) highest mode for* p/n*:*

```
p_f(M, A_1,..., A_n, H, T, D, Type, Past, Future) ←
     get_next_mode(Type, MK+1, Future, NewA, NewFuture),
     push_mode(Type, A, Past, NewPast)   |
     p_f(NewA, A_1,..., A_n, H, T, D, Type, NewPast, NewFuture).
```

Normal Clauses. The clauses for each Normal predicate are of the form c_1;;c_2. The cp(|,&) clauses in c_1 are obtained by a one-to-one translation from the source program clauses for predicate p/n in mode group *i*, as discussed above. (The three extra arguments that represent the mode-list are ignored.) c_2 consists of the following two clauses, where, again, we use clause sequencing in lieu of nested otherwise clauses.

```
p_i(A_1,..., A_n, Head, Tail, ii, _, _) ←
     p_f(NewM, A_1, ..., A_n, Head, Tail,
              ei, NewPast, NewFuture);;
 p_i(A_1,..., A_n, Head, Tail, Flag, Past, Future) ←
     copy_past(Flag, Past, Future, NewFlag, NewPast),
     get_next_mode(Flag, Last_A_Mode, Future, NewM, NewFuture);
     p_f(NewM, A_1, ..., A_n, Head, Tail,
              NewFlag, NewPast, NewFuture).
```

The first clause above is interesting. A Normal clause is called by the Entry Freeze predicate, with mode list type ii. However, the value of Future component of the mode list structure that will be passed in at run-time is known at compile time, namely, *mi*. Hence, the calls to copy_past and get_next_mode in the general clause (the second one above) may be executed at compile time. Therefore the values for the metavariables in the first clause are obtained by executing the following query at compile time:

```
← succ(mi, M1), copy_past(ii, _, M1, NewFlag, NewPast),
  get_next_mode(ii, Last_A_Mode, Future, NewM, NewFuture).
```

Here, *mi* is the *i*th mode, and *Last_A_Mode* is the (compile-time known) highest mode for the predicate.

4.5 Run time system

We omit the (straightforward) particulars of the implementation of the support predicates for mode lists, i.e. `copy_past/5, get_next_mode/5` etc.

4.5.1 Executing previously suspended goals

Once Enter Freeze or Real Freeze goals have been placed on the queue, and execution returns to the top-level, it is necessary to invoke these goals.

In [Ueda and Chikayama, 1985] this is done as follows. Suppose a goal for `p/n` has to be suspended. Place on the suspension queue the 'closure' `goal(p(A_1, ..., A_n, H, T, D), H, T, D)`. At runtime, the variables in the closure may be initialised by unifying them with actual arguments, and the closure may be called indirectly: [16]

```
execute([goal(Goal, Head, H, D) | Rest], H, T, D) ←
     incore(Goal); execute(Rest, Head, T, D).
```

The problem with this solution is that extra memory cells are used for each suspension. On a WAM-based **Prolog**, storing one goal on the suspension queue would cost $(n+11)$ words: 2 words for the cons cell on the suspension queue, 5 words for the `goal/4` structure and $(n+4)$ words for the `p/(n+3)` structure.[17] Also, the cost of one indirect invocation has to be incurred. This could be more costly in **Prolog** implementations other than **Prolog**-20, in which the more general (and slower) `call/1` mechanism has to be used.

Note that suspending and invoking a goal is a very basic step for the kinds of computations we are considering, Hence it pays to be rather careful about space and time consumption for this step.

Another solution is possible, given that *the predicates which can be placed on the suspension queue are known at compile time*. The solution corresponds to unfolding the above definition for `execute/4` one step for these known predicates, and then stripping away the unnecessary information from the suspended data-structure. Only $(n+2)$ words are needed for each suspension, and the overhead of the indirect call is avoided.

The solution is as follows. At suspension time, place the goal `p(A_1, ..., A_n, Rest)` on the suspension queue, (where the rest of the queue is `Rest`). `execute/4` will now have as many clauses as predicates for which goals can be placed on the suspension queue:

```
execute(p(A_1, ..., A_n, Rest), H, T, D) ←
     p(A_1, ..., A_n, Head, H, D); execute(Rest, Head, T, D).
```

This solution is only viable if the implementation does indexing at least on the first argument: the clauses then act as an efficient jump table for the suspended predicates. The disadvantage of this scheme

[16]**incore/1** is a **Prolog**-20 built-in predicate that takes a term and invokes the compiled predicate corresponding to the prinicipal functor of the term.

[17]I am grateful to Fernando Pereira for help with this analysis.

is that code size increases; however, code should be allocated in static space and should not interfere with the garbage collector. In fact, for some test programs, the code size actually decreased with this scheme (compared to the previous one.) Runtime decreased by about 25% and heap consumption decreased by about 20%.

A more complicated space analsysis is needed for structure-sharing implementations, such as **Prolog-20**. Even in this case, though, the jump table scheme performs better than the closure scheme.

4.6 Performance Comparisons with the Ueda/Chikayama compiler

In this section we consider various performance tradeoffs with the Ueda/Chikayama compiler (henceforth called the **Concurrent Prolog** compiler).

Unification. In our scheme, the code for checking the suspension condition for a mode group is executed at most once for every mode group, when a goal is scheduled. In the other compiler, this code is mixed in with the actual unification for every clause, and hence the same tests (for suspension) may be repeated for every clause in the mode group. Further, in their scheme it is difficult to take advantage of any head-unification optimisations (e.g. indexing) that the underlying **Prolog** may provide. '?'-unification needs to be implemented with a special purpose, complicated unification algorithm that slows down execution. In **FGHC**, terms in the head that could result in output bindings being created must be textually moved out of the head and into the body of the clause.

Multiple Modes. By keeping a representation of the modes left to evaluate for a goal, we ensure that the suspension tests for a mode succeeds at most once, and the *Real Predicate* clauses are executed at most once, regardless of the number of times the goal is suspended. This is not true for the **Concurrent Prolog** compiler. It may be the case that the blocking condition for a certain group of clauses holds, but none of the clauses is a candidate clause, and the goal has to suspend. Now each time the clause is executed from the suspension queue, these clauses will be reexamined, even though head unification will continue to fail.

Multiple mode predicates seem to arise most naturally and frequently in the context of constraint-based computation, for those predicates involved during constraint-propagation ([Saraswat, 1987c]). If blind scheduling is used, then we have discovered that a goal may be scheduled and suspended a large number of times before successfully executing. Hence it is important to restrict the amount of work done when testing a mode, if possible.

Guard execution. As discussed earlier, the **Concurrent Prolog** compiler handles full guards either inefficiently, or incorrectly, depending upon the scheduling strategy used. If depth-first scheduling is used, then it is possible that guard execution may suspend because of a *deep freeze.* In this case guard computation is failed and posssibly retried every time the goal is scheduled from the suspension queue. Worse, it is possible that the guard computation fails for legitimate reasons (i.e. there is no solution, regardless of bindings that could be generated from And-sibling goals of the parent goal). Even in this case, when the goal is rescheduled, the guard computation will be reinitiated and the failure discovered afresh.

If bounded depth first scheduling is used, then, in addition to the above problems, if the guard crosses the bound, then the implementation may never make any headway, and mistakenly report deadlock or failure.

We introduce a notion of *wait-free* guards, and use that to justify treating a deadlock inside a guard computation as a failure. Consequently a guard system is executed at most once, regardless of he number of times a goal is dequeued/enqueued from/on the suspension queue.

Suspending and Reinvoking Goals. In our scheme, exactly those goals which have been suspended need to be placed on the suspension queue, and executed via an indirect call. In the **Concurrent Prolog** scheme, *all* goals other than the first one in the body are placed on the suspension queue and invoked by an indirect call. Furthermore, we use a more space- and time-efficient way of invoking goals from the suspension queue (jump-tables vs. closures). (In principle, the jump table scheme could also be implemented in the **Concurrent Prolog** compiler.)

This results in considerable savings or those programs which have more than one goal in the body of the clause, and for which the goals do not immediately suspend due to lack of bindings. If the metaphor for computation used in writing programs is one of spawning a network of cells that subsequently receive data across links, do local computation and transmit the data across (possibly other) links, then at least the predicates that spawn the network have more than one goal in the body, and usually these goals are runnable when they are first scheduled. Indeed, if goals are runnable when first scheduled, then the performance of the cp programs is very similar to the performance of the **Prolog** programs using the same clauses (clauses in the same textual order, and goals in the body of the clause in the same textual order, and heads are identical.) The only difference is that suspension conditions are checked and some extra arguments are passed around.

We believe these issues account for the fact that execution time of the `qsort` and `sieve` benchmarks are closer to that of the **Prolog**-20 compiler than that of the Ueda/Chikayama compiler.

There are some differences in the functionality of the two compilers, that should have been apparent from the discussion. Namely, **GHC** and **Concurrent Prolog** are committed-choice CLP languages. We present an efficient compiler for a language that has a well-defined semantics for don't know non-determinsim in the context of concurrent computation. Also, as presented, our implementation is conceptually similar to a mechanism for implementing cp($\downarrow$, | ,&) in a **Prolog**-with-freeze.

4.6.1 Performance numbers

We now present detailed timing results. The following factors must be kept in mind when making the comparisons. The benchmarks run on the cp(↓, |,&) compiler and the **Concurrent Prolog** compiler are identical. Both the compilers translated into **Prolog**, which was compiled by a **Prolog-20** compiler and the resulting code run on a Dec-20 processor. However, the machines were different: hence we have included measurements of the performance of the corresponding **Prolog** programs, wherever available, on both the machines. The **Concurrent Prolog** compiler numbers have been reported in [Ueda and Chikayama, 1985].

All times are in msec.

Benchmark	Ueda/Chikayama compiler		ICOT Dec-20		CMU Dec-20
	w/o mode	w/ mode	**Prolog-20**	cp(↓, |,&) compiler	**Prolog-20**
append (500+0 elements)	79	43	12	33	13
nrev ($N = 30$)	*Not available.*			30	12
merge (100 + 100)	38	24	8	17	8
bounded buffer[a] 100 elements (size=1)	143	119	xx[b]	78	xx
bounded buffer 100 elements (size=10)	56	43	xx	41	xx
Primes (2 to 300)	886	689	188	266	222
Qsort[c] (50 elements)	119	91	17	32	21
Hanoi (N = 10)	*Not available*			63	63
Serialise[d]	*Not available*			25	10
Data Base query[e]	zz[f]			58	58

[a]A minor version of the same program, which avoided nested waits, took 29 msec and 57 msec respectively. This change should not affect the performance of the code produced by the **Concurrent Prolog** compiler.

[b]No corresponding **Prolog** program.

[c]Times are for the 50-element sequence in [Warren, 1977].

[d]Program and example as in [Warren, 1977].

[e]Program and example as in [Warren, 1977].

[f]No corresponding **Concurrent Prolog** or **GHC** program.

The programs on which the tests were run are listed in the appendix.

4.6.2 Implementation Notes

The implementation cost about 3 man weeks to develop. It consists of approximately 250 clauses, spread over approximately 80 predicates. The compiler is divided into 6 modules: the collector, the mode-analyser, the emitter, utilities and output routines, and the run-time system. The compiler released presently is Version 1.9: most of the previous versions had been concerned with examining various ways to improve the time- and space-performance of mutli-mode predicates. It is currently written in **Prolog**, but we hope to rewrite it in cp(↓, | ,&), and bootstrap in the near future.

5 Future work and conclusions

As mentioned in the introduction, we feel that this work is the first step towards the design of more direct implementations for languages in the cp family. We hope to develop ideas related to partial evaluation and program transformation rules for the basic control constructs involved so that programs may be efficiently transformed at the source level before being compiled.

The present compilation scheme also bears further study. One obvious candidate is optimising the code produced to check for the satisfaction of blocking conditions for multi-mode predicates. It seems clear that reasonable algorithms can be devised that carefully order the tests to avoid redundancies. Similarly, it seems clear that further program transformations may be done, keeping in mind the commitment to depth-first scheduling. For example, guided by information available through a static (compile time) data-flow analysis of the program, it may be possibly for some kinds of (singel mode) programs to avoid completely the overhead of `nonvar/1` and `==/1` tests at runtime, thereby generating, in effect, the directly corresponding **Prolog** programs. The programs for `qsort/2` and `nrev/2`, for example, seem susceptible to such analysis.

Acknowledgements. I acknowledge useful discussions with Ken Kahn, Seif Haridi, Mats Carlsson and Ehud Shapiro, Fred Holmgren and Annika Wærn. I am thankful to Kazunori Ueda for providing me the source code for the Ueda/Chikayama compiler, and for suggesting that we calibrate our Dec-20 machines in order to compare performances meaningfully. Many thanks to Fernando Pereira for always having the time to explain the intricacies of the **Prolog-20** implementation. Thanks to Gene Rollins for making available resources from the **Ergo** project to finish this work. This work is part of my dissertation research, which is being carried out under Prof. Dana Scott's supervision.

This work would not have been possible without a research grant from Neeta Saraswat.

A Appendix

A.1 Sample code

The following is a listing of the programs on which the benchmarks were run. Some of the programs are taken from the suite of benchmarks on which the [Ueda and Chikayama, 1985] implementation was run.

```
concat([A | X]↓, Y, [A | W])  ←  concat(X, Y, W).
concat([]↓, Y, Y).

nreverse([X | L0]↓, L)  ←  nreverse(L0, L1), concat(L1, [X], L).
nreverse([]↓, []).

merge([A | X]↓, Y, [A | W])  ←  merge(Y, X, W).
merge(X, [A | Y]↓, [A | W])  ←  merge(Y, X, W).
merge([]↓, Y, Y).
merge(X, []↓, Y).


bb(S, N) ←  buffer(S, H, T), ints(0, N, H), consume(H, T, O).

buffer(N, [_ | H1], T)  ←  N > 0, N1 is N-1 | buffer(N1, H1, T).
buffer(0, H, H).

ints(M, Max, [M | L]↓)  ←  M < Max, M1 is M+1 | ints(M1, Max, L).
ints(M, Max, [eos | _]↓)  ←  M ≥ Max | true.

consume([eos↓| Hs], [], []).
consume([H↓| Hs], [_ | Ts], [write(H), put(32), ttyflush | O2]) ←
     H\==eos | consume(Hs, Ts, O2).
```

The minor variation of the `consume/3` predicate mentioned in the benchmark table is the program:

```
consume([H | Rest], Tail, Output) ←
     consume(H, Rest, Tail, Output).
 consume(H↓, [NewH | Rest], Tail,
         [write(H),put(32),ttyflush|O]
        )←
      H\==eos | consume(NewH, Rest, Tail, O).
 consume(eos↓, Rest, [], []).
```

In principle this program could be obtained automatically from the program above, by means of some simple unfolding/folding techniques, directed by the knowledge that it is useful to replace nested waits by top-level waits, at the expense of introducing extra arguments.

The other programs are:

Sieve of Eratosthenes:

```
gen(N, Max, [N | S1])  ←  N < Max, M is N+1 | gen(M, Max, S1).
gen(N, Max, [])  ←  N ≥ Max | true.

sift([P|L]↓, [write(P), put(32), ttyflush | S1]) ←
     filter(L, P, K), sift(K, S1).
 sift([]↓, []).

filter([Q | L]↓, P, K)  ←  (Q mod P) =:= 0 | filter(L, P, K).
filter([Q | L]↓, P, [Q | K1])  ←  Q mod P ≠ 0 | filter(L, P, K1).
filter([]↓, P, []).
```

Quick Sorting:

```
qsort(Xs, Ys)  ←  qsort(Xs, Ys, []).

qsort([X | Xs]↓, Ys0, Ys2)  ←
     part(Xs, X, S, L), qsort(S, Ys0, [X|Ys1]), qsort(L, Ys1, Ys2).
 qsort([]↓, Y, Y).

part([X | Xs]↓, A, S, [X | L1])  ←  A < X | part(Xs, A, S, L1).
part([X | Xs]↓, A, [X | S1], L)  ←  A >= X | part(Xs, A, S1, L).
part([]↓, A, [], []).
```

A.1.1 Some other benchmarks

The hanoi/2 program is a cp(|) program: the translated program produced is a syntactic variant with a cut in place of '|'.

A cp(|) *program for the Towers of Hanoi:*

```
hanoi(N, Moves) ←  hanoi(N, a, b, c, Moves, stop).

hanoi(1, A, B, C, move(A,B, M), M).
hanoi(N, A, B, C, Move, Rest) ←
     N ≥ 1, N1 is N-1 |
     hanoi(N1, A, C, B, Move, M1),
     hanoi(1, A, B, C, M1, M2),
     hanoi(N1, C, B, A, M2, Rest).
```

The following benchmark programs are taken from [Warren, 1977].

Serialising a List:

```
serialise(L, R) ←
     pairlists(L, R, A), arrange(A, T), numbered(T, 1, N).

pairlists([X | L]↓, [Y | R], [pair(X,Y) | A]) ←
     pairlists(L, R, A).
 pairlists([]↓, [], []).

arrange([X | L]↓, tree(T1, X, T2)) ←
     split(L, X, L1, L2), arrange(L1, T1), arrange(L2, T2).
 arrange([]↓, void).

split([X | L]↓, X, L1, L2) ←
     split(L, X, L1, L2).
 split([X | L]↓, Y, [X | L1], L2) ←
     before(X, Y) | split(L, Y, L1, L2).
 split([X | L]↓, Y, L1, [X | L2]) ←
     before(Y, X) | split(L, Y, L1, L2).
 split([]↓, Y, [], []).

before(pair(X1, Y1), pair(X2, Y2)) ←  X1 < X2.

numbered(tree(T1, pair(X, N1), T2)↓, N0, N) ←
     numbered(T1, N0, N1), inc(N1, N2),
     numbered(T2, N2, N).
 numbered(void↓, N, N).

inc(N↓, N1) ←  N1 is N+1.
```

A.1.2 Data base query

This program uses the sequencing obtained by commitment in order to ensure that bindings for variables occuring in arithmetic goals are available before the goals are executed. Note that each of the predicates in the program is wait-free, and that the guards contain calls only to wait-free predicates. Hence this program is translated by the cp(↓, | ,&) compiler into the **Prolog** program presented in [Warren, 1977].

```
query(X) ←
     density(C1, D1), density(C2, D2) & D1 > D2, 20 × D1 < 21 × D2.
 density(C, D) ←
     pop(C, P), area(C, A) & D is (P *100)/A.
```

The database used for the query is retained from Warren's thesis, even though it is hopelessly out of date. We omit listing it here.

In reality a call to query/1 should occur in the guard of a clause so that top-level computation is not split unless a solution to the query is obtained. Hence for efficiency in a parallel implementation, the following clause may be needed:

```
query2(X) ←  query(X) &true.
```

with all calls to `query/1` in the program being replaced by calls to `query2/1`. (This additional clause does not affect the performance of the code produced by the cp(↓, | ,&) compiler.)

References

[Boizumault, 1986] P. Boizumault. A general model to implement dif and freeze. In *Proceeedings of the 3d International Conference on Logic Programming, London*, pages 585–592, May 1986.

[Carlsson, 1987] Mats Carlsson. Freeze, indexing and other implementation issues in the wam. In *Proceeedings of the 4th International Conference on Logic Programming, Melbourne*, pages 40–58, May 1987.

[Clark and Gregory, 1985] Keith Clark and Steve Gregory. Notes on the implementation of **Parlog**. *J. of Logic Programming*, 1:17–42, 1985.

[Codish, 1985] Michael Codish. *Compiling Or-parallelism into And-parallelism.* Technical Report, Weizmann Institute, Israel, December 1985. MSc thesis.

[Cohen, 1985] Jacques Cohen. Describing **Prolog** by its interpretation and compilation. *Communications of ACM*, (12), December 1985.

[Courcelle, 1983] B. Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Scienc*, 25():95–169, 1983.

[Gregory, 1984] S. Gregory. *Implementing PARLOG on the abstract PROLOG machine.* Technical Report Res report DOC 84/23, Imperial College, August 1984.

[Gregory, 1987] Steve Gregory. *Parallel Logic Programming in* **Parlog**. *International Series in Logic Programming*, Addison-Wesley, 1987.

[Holmgren and Wærn, 1987] Fredrik Holmgren and Annika Wærn. *A scheme for compiling* **GHC** *to* **Prolog** *using freeze.* Technical Report SICS R87007, Swedish Institute for Computer Science, 1987.

[Kliger, 1987] Shmuel Kliger. *Master's thesis.* Technical Report, Weizmann Institute, Israel, 1987.

[Miyazaki *et al.*, 1985] T. Miyazaki, A. Takeuchi, and T. Chikayama. A sequential implementation of **Concurrent Prolog** based on the shallow binding scheme. In *Symposium on Logic Programming*, IEEE, July 1985.

[Saraswat, 1985] Vijay A. Saraswat. Partial correctness semantics for cp(↓, | ,&). *FSTTCS Conference proceedings*, (206):347–368, 1985.

[Saraswat, 1986] Vijay A. Saraswat. *Problems with* **Concurrent Prolog**. Technical Report 86-100, CMU, January 1986.

[Saraswat, 1987a] Vijay A. Saraswat. Detecting distributed termination efficiently: the short-circuit technique in fcp(↓, |). February 1987. To be submitted.

[Saraswat, 1987b] Vijay A. Saraswat. The concurrent logic programming language cp: definition and operational semantics. In *Proceedings of the SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 49–62, ACM, January 1987.

[Saraswat, 1987c] Vijay A. Saraswat. cp as a general-purpose constraint language. In *Proceedings of the National Conference, Seattle*, American Association for Artificial Intelligence, July 1987.

[Saraswat, 1987d] Vijay A. Saraswat. cp as a logic programming language. August 1987. Treats the downward completeness results for CP.

[Saraswat, 1987e] Vijay A. Saraswat. The language **GHC**: operational semantics and comparison with cp(↓, |). In *IEEE SLP Proceedings, San Francisco*, September 1987.

[Saraswat, 1987, Forthcoming] V. A. Saraswat. Merging many streams efficiently: the importance of atomic commitment. In Ehud Y. Shapiro, editor, **Concurrent Prolog** *Collected Papers*, MIT Press, 1987, Forthcoming.

[Saraswat, forthcoming] Vijay A. Saraswat. *Concurrent Logic Programming Languages.* PhD thesis, Carnegie-Mellon University, forthcoming.

[Shapiro, 1983] E.Y. Shapiro. *A subset of* **Concurrent Prolog** *and its interpreter.* Technical Report CS83-06, Weizmann Institute, 1983.

[Shapiro, 1987] Ehud Y. Shapiro. *Implementing* **Prolog** *on* **Concurrent Prolog** *(??)* In *Proceedings of the Fourth International Logic Programming Conference*, May 1987.

[Silverman, 1986] Silverman. *Logix system.* Department of Computer Science, Weizmann Institute, Israel, 1986.

[Takeuchi and Furukawa, 1986] Takeuchi and Furukawa. Concurrent logic programming languages. In *Proceedings of the Third International Conference on Logic Programming*, July 1986.

[Ueda, 1986] Kazunori Ueda. Making exhaustive search programs deterministic. In *Proceedings of the Third Internation Logic Programming Conference*, July 1986. Also in New Generation Computing, 1987.

[Ueda and Chikayama, 1985] K. Ueda and T. Chikayama. A compiler for **Concurrent Prolog**. In *Proceedings of the second International Symposium on Logic Programming*, IEEE, July 1985.

[Warren, 1977] D.H.D. Warren. *Applied Logic — Its use and implementation as a programming tool.* PhD thesis, University of Edinburgh, 1977.

[Warren, 1987] D.H.D. Warren. The SRI model for or-parallel execution of **Prolog** — abstract design and implementation issues. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 92 – 103, IEEE, August 1987.