A Parallel Implementation of the Andorra Kernel Language

Douglas Frank Palmer

Technical Report TR 97/21

June, 1997

Submitted in total fulfilment of the requirements
of the degree of Doctor of Philosophy

Department of Computer Science
University of Melbourne
Parkville, Victoria
Australia

Abstract

The Andorra Kernel Language (AKL), also known as the Agents Kernel Language, is a logic programming language that combines both don't know nondeterminism and stream programming.

This thesis reports on the design and construction of an abstract machine, the DAM, for the parallel execution of AKL programs. Elements of a compiler for the DAM are also described.

As part of the development of the DAM, a bottom-up abstract interpretation for AKL and a logic semantics for the AKL, based on interlaced bilattices have also been developed. This thesis reports on the abstract interpretation and the logical semantics.

This thesis is less than 100,000 words in length, exclusive of tables, bibliography and appendices.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank my supervisor, Dr. Lee Naish for his patience and support. My discussions with him were always both entertaining and informative. I would particularly like to thank him for his tolerance of my tendency to vanish and reappear at odd intervals. Thanks also go to the other two members of my supervisory committee, Harald Søndergaard and Zoltan Somogyi.

Great thanks, also go to Alison Wain, my wife, who was convinced that this was what I *should* be doing. Without her moral and financial support, this thesis would never have seen the light of day.

The Department of Computer Science at Melbourne University has always been a friendly and supportive environment. Particular thanks go to my room-mates, Fergus Henderson, Robert Holt, Peter Schachte, and the "dinner circle" of Chaoyi Pang, Devindra Weerasooriya and Eric Yeo, for their stimulating discussions.

Thanks also go to the various lecturers and tutors who have used me as a casual tutor over the years. Especially Linda Stern, Harald Søndergaard and Rex Harris; the best way to learn something is to teach it. To the undergraduates and diploma students: well, it had to be someone.

The management of Applied Financial Services were flexible and supportive while I was writing up. Thankyou to them.

While studying, I was supported by a Commonwealth Postgraduate Research Allowance and later an Australian Postgraduate Research Award.

— - Uses LaTex

Chapter 1

Introduction

Logic Programming [Kow74] and its practical realisation in Prolog [Rou75] introduced a new paradigm to computer science. Logic programming has a declarative model, where programs are represented by relationships between entities, rather than by instructions on how to solve problems (the imperative model). Prolog and logic programming have found ready use in areas where problems can be understood in terms of relationships between interacting parts: expert systems, natural language recognition, theorem proving.

An example logic program (in Prolog), which can be used to find paths through a graph, is:

```
arc(a, b).
arc(b, c).    arc(b, d).
arc(c, e).    arc(c, g).
arc(d, f).    arc(d, g).
arc(e, g).

path(X, Y) :- arc(X, Y).
path(X, Y) :- arc(X, Z), path(Z, Y).
```

Each statement in the program is termed a *clause*. Groups of clauses, with the same name and number of arguments form a *predicate*. Predicates are normally referred to as $name/args$, eg. `path/2`.

The program consists of a database of facts, the `arc/2` predicate, and a means of constructing paths, the `path/2` predicate. In English, `path/2` can be read as "there is a path from X to Y if there is an arc from X to Y, also there is a path from X to Y if there is an arc to some intermediate location, Z, and a path from Z to Y."

This program can be queried by giving it a goal, such as `?-path(a, f)`, which can be interpreted as "is there a path from `a` to `f`?" A Prolog interpreter essentially acts as a theorem-prover, attempting to find a proof for the goal. Clearly, there is some trial-and-error involved and one of the most interesting aspects of Prolog, and logic programming in general, is its inherent nondeterminism. In searching for a path from `a` to `f`, the Prolog interpreter will attempt to construct a series of arcs $\texttt{a} \rightarrow \texttt{b}, \texttt{b} \rightarrow \texttt{c}, \texttt{c} \rightarrow \texttt{e}, \texttt{e} \rightarrow \texttt{g}$. At this point, there are no arcs which lead out from `g`, and the Prolog interpreter is unable to satisfy either part of the path definition. The interpreter must backtrack to a suitable point, and attempt to construct an alternate route to `f`; in this case $\texttt{a} \rightarrow \texttt{b}, \texttt{b} \rightarrow \texttt{d}, \texttt{d} \rightarrow \texttt{f}$.

As an alternative, the program can be interrogated with a query such as `?-path(c, X)`, which can be interpreted as "what nodes can be reached from `c`?" A Prolog interpreter will construct the first available solution from the definition of `path/2` and return with the answer `X = d`. If another answer is requested, then the interpreter backtracks to produce `X = g` and `X = g` again (derived from the path $\texttt{c} \rightarrow \texttt{e}, \texttt{e} \rightarrow \texttt{g}$). Similarly, a goal such as `?-path(X, f)` will give all the nodes that can reach `f`. The declarative programming of `path/2` allows it to be used for several different purposes, purposes which would have to be explicitly programmed into imperative languages.

Declarative programming also allows a certain amount of order independence in its definitions. For example, in the `path(X, Y) :- arc(X, Z), path(Z, Y)` clause, there is no reason why the

`arc(X, Z)` part must be evaluated before the `path(Z, Y)` part. Although standard Prolog always evaluates parts of a body in strict left to right order, more advanced versions of Prolog, such as NU-Prolog [ZT86] or SICStus Prolog [SIC88] allow a user-defined order of evaluation.

Rather than order independence, parts of goals can be evaluated in parallel, giving Prolog an inherently parallel character. There are essentially two forms or parallelism extractable from Prolog programs: And-parallelism attempts to evaluate individual clauses in parallel; Or-parallelism attempts to parallelise the nondeterminate matching of clauses, evaluating several possible branches simultaneously.

The introduction of parallelism into Prolog introduces several difficulties, especially in the case of and-parallelism. Variables in Prolog are single assignment variables; once given a value, the variable does not change. Single assignment variables are similar to variables as used in mathematics, representing a common value at all points where they are used. In the case of `arc(X, Z), path(Z, Y)` the `Z` variable is shared by both parts of the clause. If both parts are run in parallel, then some means of synchronising the two parts must be found.

A huge variety of attempts to solve the various problems of parallelism in logic programming have been made over the years. The Andorra/Agents Kernel Language (AKL) [Jan94] is an attempt to unify many of these attempts, as well as provide a general formal structure for handling logic programming. This thesis presents an implementation of the AKL, designed for parallel execution.

1.1 Thesis Outline

This thesis is a report on the implementation of a parallel abstract machine for the AKL.

Chapter 2 provides an introduction to the various forms of parallelism that logic programming languages are capable of. Chapter 3 is a description of the Andorra model and the AKL.

The basic motivation behind the thesis is the design of an abstract machine, the DAM, for the parallel execution of AKL programs. A description of the DAM can be found in chapter 5.

A compiler for the abstract machine is discussed in chapter 6. Parts of the DAM can be expensive to execute, especially the machinery that is used to handle nondeterminism. The compiler uses an abstract interpretation to gather data about the entire program before performing the compilation, enabling more efficient ordering of the goals within clauses, and the early selection of determinate clauses.

The abstract interpretation uses a logical semantics for the AKL based on bilattices. Bilattices allow an extension to the normal two-valued Boolean logic that can capture the more complex behaviour of the AKL. This logical semantics is described in chapter 4, and soundness and completeness theorems are provided for the AKL. Despite being a by-product of the attempt to produce the DAM, this semantics is probably the most interesting aspect of this thesis.

Original contributions in this thesis are the concept of variable localisation in the DAM, the use of bit-mapped clause sets for clause indexing, the broad type abstract domain and the bilattice formulation of the AKL's logical semantics.

1.2 Some Preliminaries

This section is intended to provide a convenient reference to the standard terminology used to describe logic programs. Most of this terminology is derived from Lloyd [Llo84].

1.2.1 First Order Logic

Most logic programming has first order logic as a foundation. This section provides an informal guide to the terminology of first order logic.

First order theories are built from *variables*, *constants* and *function* and *predicate* symbols. Functions and predicates have an *arity*, which is the number of arguments that they take. A *term* is defined recursively as: a constant is a term and a variable is a term; if f is a function with arity n and $t_1, \ldots, t_n$ are terms then $f(t_1, \ldots, t_n)$ is a term. A *ground* term contains no variables.

An *atom* $p(t_1, \ldots, t_n)$ is constructed from a predicate p with arity n and the terms $t_1, \ldots, t_n$. A *formula* is defined recursively by A, $\neg F$, $F \wedge G$, $F \vee G$, $F \leftarrow G$, $F \leftrightarrow G$, $\exists x F$ and $\forall x G$ where A is an atom, x is

a variable and F and G are formulae. The meaning of the conjunctions is: $\neg$ is negation, $\wedge$ is conjunction (and), $\vee$ is disjunction (or), $\leftarrow$ is implication and $\leftrightarrow$ is equivalence. The expression $\bigvee_{s \in S} F(s)$ means $F(s_1) \vee \cdots \vee F(s_n)$ for all $s \in S$, similarly for $\bigwedge_{s \in S} F(s)$. $\exists x F$ means there exists an x for which F is true. $\forall x F$ means that F is true for all x. A formula is *closed* if all variables are quantified by $\exists$ or $\forall$. By an abuse of notation $\exists F$ or $\forall F$ can be taken to mean that F is quantified over all variables that occur F. An atom, or the negation of an atom is called a *literal*. A *clause* is a formula of the form $\forall x_1 \cdots \forall x_p (A_1 \vee \cdots \vee A_n \leftarrow B_1 \wedge \cdots \wedge B_m)$. A *definite clause* or *program clause* has the form $\forall x_1 \cdots \forall x_p (A \leftarrow B_1 \wedge \cdots \wedge B_m)$.

An *interpretation* consists of the domain of the interpretation (D), an assignment of an element of D to each constant in the theory, an assignment of an element of D to each D^n for each function of arity n in the theory and an assignment of either true or false to each D^n for each predicate of arity n. A *Herbrand interpretation* simply has a domain of all the constants and functions in the theory and interprets a constant c as c and a function $f(t_1, \ldots, t_n)$ as $f(t_1, \ldots, t_n)$.

An interpretation I is a *model* for a set of closed formulas S if applying the values of the interpretation to each $F \in S$ results in F evaluating to true. I models S is denoted by $I \models S$. A *Herbrand model* for S is a Herbrand interpretation that models S. Herbrand models have the convenient property that sets of clauses are only unsatisfiable if they have no Herbrand models.

1.2.2 Prolog

Prolog is the original logic programming language. Most other logic programming languages introduce further elements of syntax and execution model to the common Prolog base.

Constants in Prolog are represented by initial lower case letters or numbers, eg. `foo` or `2.2`. Functions are represented by a lower case functor, and a sequence of arguments in parentheses, eg. `f(a, b)`; some function symbols can be written as infix operators, eg. `A + B` is equivalent to `+(A, B)`. Variables start with upper case letters or underscores, eg. `X` or `_`. Variables starting with underscores are anonymous variables, each different from the other. Lists are denoted by `[`$e_1, \ldots, e_n$`]`, with the $e_1, \ldots, e_n$ being the elements of the list, eg. `[1, 2, 4, 8]`. The construction `[`$e_1, \ldots, e_m$ `|` T`]`, is a partial list, where $e_1, \ldots, e_m$ comprise the head elements of the list and T is the tail, eg. `[push(X) | R]`.

Clauses are written as H `:-` $A_1, \ldots, A_n$ where H is the head of the clause and $A_1, \ldots, A_n$ is the clause body, with each A_i a literal. A clause with no literals is called a *fact*. The normal logical meaning for a clause is $\forall X H \leftarrow \exists Y (A_1 \wedge \cdots \wedge A_n)$ where X is the set of all the variables that occur in the head, and Y is the set of all the variables that appear in the body only.

A *substitution* is a mapping from variables to terms, written as $\{V_1/T_1, \ldots, V_n/T_n\}$, where V_i is a variable and T_i is a term. If a substitution θ is applied to a term T, written as $T\theta$ then all instances of variables in θ which are found in T are replaced by the corresponding term. Eg. $f(A, g(X, Y))\{A/f(a, Y), X/Y\} = f(f(a, Y), g(X, Y))$. Substitutions can be composed to form other substitutions, with the composition of θ and σ written as $\theta\sigma$. A variable in a substitution is *bound*.

A substitution θ is a *unifier* for a set of terms T if $\{T_i\theta : T_i \in T\}$ is a singleton. Eg. $\{A/a, B/c\}$ is a unifier for $\{f(A, c), f(a, B), f(A, B)\}$. The *most general unifier* for a set of terms T, written as mgu(T) is the substitution θ such that all other unifiers of T can be composed from θ and some other substitution ϕ; $\theta\phi = \sigma$. The *unification* of two terms T and S is the computation of mgu(T, S).

A Prolog program is evaluated by means of *SLD-Resolution*. A *goal* consists of a sequence of literals $G_1, \ldots, G_n$. Each step in SLD-Resolution consists of selecting a literal from the goal, G_i and finding a clause H `:-` $A_i, \ldots, A_m$ where G_i and H are unifiable, with most general unifier θ. The new goal is then $(G_1, \ldots, G_{i-1}, A_1, \ldots, A_m, G_{i+1}, \ldots, G_n)\theta$. If the goal eventually dwindles to an empty list, then θ is an *answer substitution* for $G_1, \ldots, G_n$.

The function which decides which G_i to select for expansion is called the *computation rule*. Prolog uses a computation rule that always selects the left-most literal. A computation rule is *fair* if a literal will always eventually be selected; the Prolog computation rule is not fair.

An SLD-Resolution fails when there are no clauses that match the selected atom. In such a case, the computation *backtracks*: it backs up a step, and selects an alternate clause to try. If no alternate clause exists, then the computation backs up another step, until a new clause is found, or until all steps are eliminated and the entire computation fails. The order in which clauses are selected is called the *selection rule*. Prolog

uses a strict top-to-bottom selection rule. As forward execution and backtracking alternate, the computation builds a tree called an *SLD-tree*.

An attraction of SLD-Resolution is that it can be shown to correctly compute all the answer substitutions for a goal. SLD-Resolution is *sound* in the sense that any computed answer substitution is logically implied by the program. SLD-Resolution with a fair computation rule is *complete* in the sense that any possible correct answer substitution is always eventually computed.

The *cut* allows pruning within a Prolog program and is added to a clause by

$$H \quad \texttt{:-} \quad A_1, \ldots A_i, \ \texttt{!} \quad , A_{i+1}, \ldots, A_n$$

If all the literals to the left of the cut have been eliminated, then the cut prunes the SLD-tree, removing any alternate clauses which could be found for $A_1, \ldots, A_{i+1}$ and any alternate clauses for H.

1.2.3 Constraints

Prolog essentially uses a Herbrand interpretation, with a few concessions to arithmetic, to decide whether a goal is satisfiable. Substitutions and unification allow an answer to be computed. However, logic programming can be extended to cover a wider range of interpretations. *Constraint Logic Programming* [JL87] extends logic programming to handle a variety of constraint systems, where constraints can be arbitrary closed formulae built from primitive predicates. A *constraint theory* is an interpretation of the constraint domain. A constraint θ is satisfiable in a constraint theory C if $C \models \theta$. A constraint θ *entails* another constraint σ if $C \models \theta \rightarrow \sigma$.

Substitutions can be seen to be a special kind of constraint, with $\{V_1/t_1, \ldots, V_n/t_n\}$ being replaced by the constraint $\{V_1 = t_1 \wedge \cdots \wedge V_n = t_n\}$. The constraint theory of *Herbrand equality* interprets the equality predicate as equality on Herbrand terms.

1.2.4 Lattices

Lattices are a generalisation of ordered sets and are useful in describing the logical semantics of logic programming. This characterisation of lattices is taken from [Llo84].

A relation R on a set S is a *partial order* if xRx, $xRy \wedge yRx \rightarrow x = y$ and $xRy \wedge yRz \rightarrow xRz$ for all $x, y, z \in S$. A *poset* is a set with some partial ordering.

If S is a poset with partial order $\leq$ then a is an *upper bound* of $X \subseteq S$ if $x \leq a$ for all $x \in X$. Similarly, a is a *lower bound* of X if $a \leq x$ for all $x \in X$. X may not always have an upper or lower bound, depending on the nature of the poset. The *least upper bound* of X is the smallest possible upper bound on X and is denoted by $\text{lub}(X)$. The *greatest lower bound* of X is the largest possible lower bound of X and is denoted by $\text{glb}(X)$.

A poset L is a *complete lattice* if $\text{lub}(X)$ and $\text{glb}(X)$ exist for all $X \subseteq L$. A complete lattice has a *top element*, $\text{lub}(L)$, denoted by $\top$ and a *bottom element*, $\text{glb}(L)$, denoted by $\bot$.

A mapping $T : L \rightarrow L$ is *monotonic* if $x \leq y \rightarrow T(x) \leq T(y)$ for all $x, y \in L$. A *fixpoint* of T is an element $a \in L$ where $T(a) = a$. The *least fixpoint* of T is defined as $\text{lfp}(T) = \text{glb}(\{x : T(x) = x\})$. Similarly, the *greatest fixpoint* of T is defined as $\text{gfp}(T) = \text{lub}(\{x : T(x) = x\})$.

$X \subseteq L$ is *directed* if every finite subset of X has an upper bound in X. T is *continuous* if $T(\text{lub}(X)) = \text{lub}(T(X))$ for every directed subset X of L.

Chapter 2

An Overview of Parallel Logic Programming

This chapter presents a general overview of the bewildering variety of parallel logic programming systems, with the exception of those based on the Andorra principle, which are discussed in chapter 3.

Most parallel logic programming systems are based on the familiar Prolog and attempt to provide a degree of transparent parallelism. In principle, there are two basic forms of parallelism which can be exploited in logic programs. *Or-Parallelism* attempts to derive several answers to a non-determinate goal in parallel. *And-Parallelism* attempts to execute several parts of a goal in parallel. In turn, and-parallelism can take two subsidiary forms: *Independent and-parallelism* evaluates conjunctions that are independent of each other (ie. no shared variables). *Dependent and-parallelism* evaluates conjunctions that share information.

Most models of parallelism in logic programming languages view the computation as an and-or tree [Con83]. The computation tree consists of alternating layers of and- and or-nodes. Conjunctions of goals running in parallel are viewed as and-nodes. Disjunctions of possible answers are regarded as or-nodes, with each or-branch representing a choice.

2.1 Or-Parallelism

Or-Parallelism normally takes a logic program and attempts to transparently evaluate successive or-branches in the computation tree in parallel.

An example of a program where or-parallelism can be exploited is the traditional ancestor program, shown in figure 2.1. The query `?-ancestor(cedric, gustavus)` can proceed in parallel as each call to `parent(X, Z)` produces a new crop of possibilities.

The normal view of or-parallelism is that of several processors, called *workers*, standing ready to explore or-branches. An or-parallel computation normally proceeds by evaluating a query until a nondeterminate call is reached. If there are idle workers, several clauses can be evaluated in parallel. When clauses are evaluated in parallel, multiple bindings may be made to a single variable. The essential problem in or-parallelism is how to resolve the multiple binding problem.

2.1.1 The Hash Window Binding Model

The hash window binding model was developed by Borgwardt [Bor84] and is used in the Argonne National Laboratory's parallel Prolog [BDL+88] and the PEPSys system [WR87]. Each alternative or-branch maintains a hash table, called a hash window, for storing conditional bindings. When a process makes a binding to a variable that other processes may be able to bind to, the binding is stored in a hash window. Dereferencing a variable involves searching up through the chain of hash windows until a binding is found. The hash window model differs from the other models presented below (sections 2.1.2, 2.1.3 and 2.1.4) in that the extra data structures used to handle multiple bindings are associated with the search-tree rather than with the worker.

```
parent(jemima, rose).
parent(jemima, bill).
parent(cedric, rose).
parent(cedric, bill).
parent(cedric, alonzo).
parent(betty, alonzo).
parent(rose, fredrick).
parent(rose, david).
parent(mark, fredrick).
parent(mark, david).
parent(bill, peter).
parent(betty, peter).
parent(fredrick, gustavus).

ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).

?-ancestor(cedric, gustavus).
```

Figure 2.1: An Example Program for Or-Parallelism

By itself the hash window system is very inefficient, as every dereference to a possibly shared variable needs to work through the ascending chain of hash windows. The shallow binding [BDL+88, WR87] optimisation reduces the need for entries in hash windows. Once a workers starts on a branch of the tree, any variables that it creates and then binds need not be entered into the hash window, as the variable is only visible to the creating worker. The worker can trail variables and backtrack, provided that it does not export any or-branches to another worker.

Scheduling using hash windows is very flexible. A context switch, where a worker switches to an unexplored part of the computation tree simply involves changing the hash table that the worker uses.

A computation for the example from figure 2.1 using hash windows is shown in figure 2.2. This computation has two workers, which are currently exploring alternate branches in the or-tree.

2.1.2 The Binding Array Model

The binding array model has been used in both one version of the SRI-Model [War87] and the Aurora system [LBD+90]. Each worker maintains an array of bindings, with shared variables having the same index into the array across workers.

When a variable that has not been conditionally bound is conditionally bound, a new binding entry is added to the top of the array, and the variable is associated with the entry. Bindings are stored in the associated array entry for that worker. Two workers share the same bindings to the extent that their binding arrays contain the same entries. Each or-node stores the current top of the array. Another worker can acquire an or-branch from a worker by synchronising its bindings up to the index maintained in the or-node. An example of the binding array model, using the example from figure 2.1 and three workers is shown in figure 2.3.

When a worker finishes a branch of the computation tree, it needs to be rescheduled to work on another branch of the tree. Moving to another branch of the tree involves unwinding the binding array up to the shared or-node between the original and new branches, and then acquiring the new binding array from the new branch. Optimal scheduling for the binding array model, therefore, means that a worker has to move as small a distance as possible from its original position in the tree, to avoid the copying overhead of a large move.

The Manchester scheduler [CS89] keeps two global arrays, indexed by worker number. The first array contains the tasks that each worker has available for sharing, along with information on how far the task

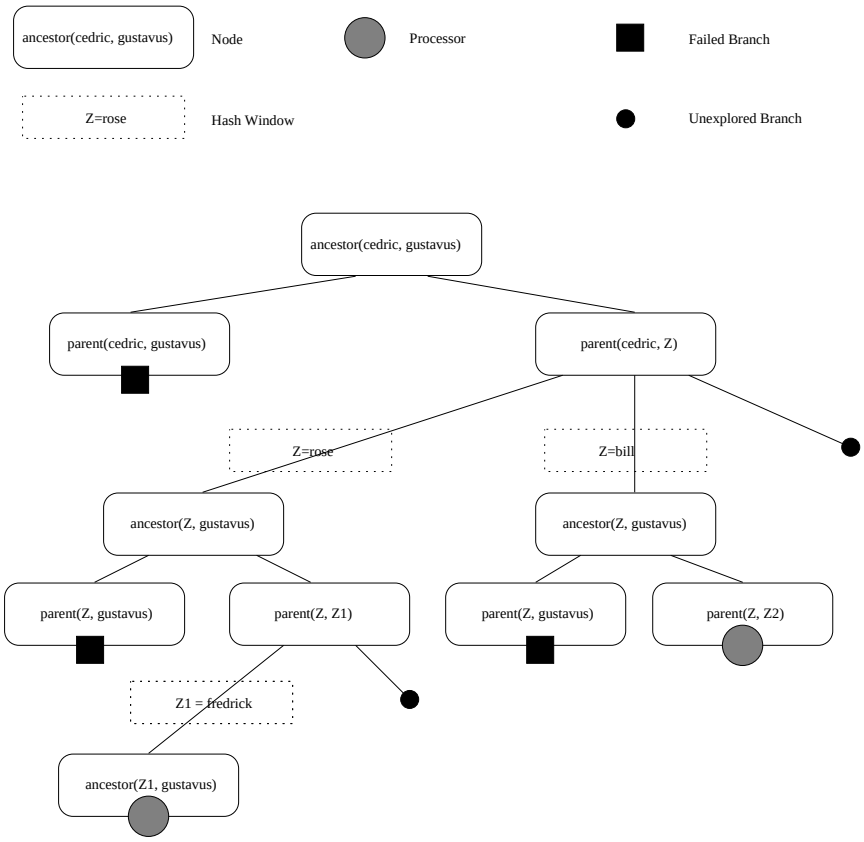

Figure 2.2: Example Or-Parallelism Using Hash Tables

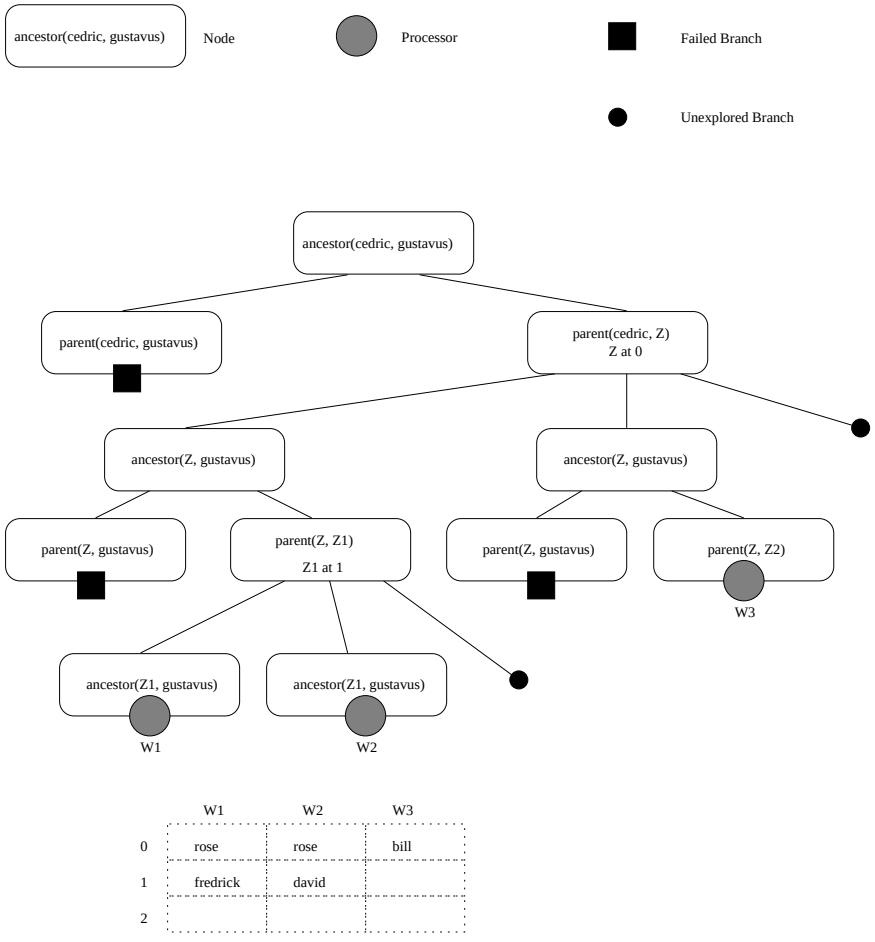

Figure 2.3: Example Or-Parallelism Using Binding Arrays

is from the root of the computation tree. The second array contains information about the status of each worker, and how far it is from the root if idle. When a worker becomes idle, it is assigned the task with the least migration cost. If there are no available tasks, workers shadow active workers; when work becomes available, the shadowing workers can cheaply pick the work up.

Binding arrays provide a more complex scheduling problem than hash windows, which can be rapidly moved to any unexplored branch. Binding arrays, however, provide the advantages of constant access time to bindings. Intuitively, binding arrays should be superior to hash windows in cases where there are few workers, leading to fewer large context changes.

2.1.3 The Multi-Sequential Machine Model

The multi-sequential machine model was proposed separately by Ali [Ali86] as the multi-sequential machine and Clocksin (further refined by Alshawi) as Delphi [AM88], Both models are designed to allow or-parallelism with a minimal amount of communication between processes. Clever initialisation of workers allow workers to distribute or-branches between themselves with little communication.

The multi-sequential machine model starts a number of workers executing the same program. Each worker is given a virtual worker number and the number of worker within its group. When an or-branch is reached, the workers split the work amongst themselves; each worker knows how many workers are in the group, and what its worker number is, so the workers can reach agreement on which branches to take without communication. As an example, suppose that 5 workers reach a three way branch, then two workers could be assigned to the first branch (to further split when encountering another branch), two to the second and one to the third. Balanced, left-biased and right-biased allocation schemes are all possible allocation strategies.

When a worker becomes idle after completing all solutions, it is assigned to a local manager. The local manager collects a group of idle workers and, when the group is large enough, requests work from a busy worker group. The state of the busy worker group is copied to the new group and then the new group is started as an independent worker group.

The Delphi model is designed to avoid workers having to exchange state. The model uses bit strings, called *oracles* to control the search space, which has been pre-processed into a binary search tree. A central manager sends a worker an oracle, giving a path to search. The path is searched to a given depth, and then either solutions, failure or additional oracles are returned to the manager. As the program executes, the oracles grow to represent deeper and deeper branches.

When a worker receives an oracle, it can re-synchronise itself by backtracking along its current oracle until the two oracles are the same and then following the path of the new oracle to where processing has started. Following the new path may be expensive.

2.1.4 The Copying Model

The copying model was proposed by Ali [AK90] for the Muse or-parallel Prolog system. The copying model, rather than trying to maintain shared bindings for the same variable uses copying of the entire worker's workspace to allow multiple bindings.

Copying is similar to the binding array model, except that all the workspace is synchronised rather than just the variable bindings. When an or-branch becomes available for parallel execution, another worker can acquire the or-branch by asking for a copy of the stacks used in the computation. Only the parts of the stacks that differ between the two workers need to be copied. The worker acquiring the work is then in the same state as the original worker; it can then backtrack and take the next available or-branch.

Although complete copying is expensive, the corresponding advantage to using copying is that, once copying has finished, each worker is largely independent of all other workers and can dereference and bind variables without any overhead. Performance results suggest that the copying method is often superior to the binding array method [AK91b].

```
diff(c, 0).
diff(x, 1).
diff(A + B, DA + DB) :-
       diff(A, DA),
       diff(B, DB).
diff(A * B, (DA * B) + (A * DB)) :-
       diff(A, DA),
       diff(B, DB).

?-diff((x * 5) + (2 * x), D).
```

Figure 2.4: An Example Independent And-Parallel Program

```
qsort([P | U], S) :-
       partition(P, U, U1, U2),
       qsort(U1, S1),
       qsort(U2, S2),
       append(S1, S2, S).

?-qsort([5, 6, 2, 1, 9, 0, 3], S).
```

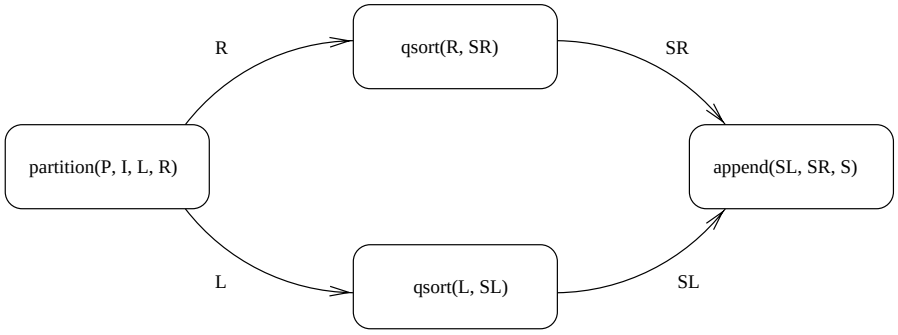

Figure 2.5: Example Independent And-Parallelism Call-Graph

2.2 Independent And-Parallelism

Independent and-parallelism (IAP) attempts to transparently exploit the parallelism which appears when two goals in a conjunction have no common variables. If the goals share no variables, then the two goals can be evaluated in parallel without the need for any synchronisation between processes.

An example program, a fragment of a differentiator, which can be run with IAP is shown in figure 2.4. The clauses that handle compound expressions each call `diff/2` twice recursively. If the expressions are independent of each other (ground, or no shared variables) then the recursive differentiations can be run in parallel. If the recursive differentiations do contain shared variables (eg. `?-diff(X * X, D)`) then the recursive differentiations must be run in sequence.

The central problem in IAP is the construction of a call-graph showing which literals in a clause must be run in sequence and which can be run in parallel. An example call-graph is shown in figure 2.5. The largest difference in implementations of IAP is whether the call-graphs are constructed by run-time checks, static analysis at compile-time, or by some hybrid of the two.

2.2.1 Run-Time Detection of Independent And-Parallelism

The flow of dependency between subgoals can be detected at run-time and the call-graph adjusted dynamically.

An example of the run-time approach is Lin and Kumar's bit-vector model [LK88]. The bit vector model essentially associates a token with each variable in a clause. The tokens are passed from literal to literal, with a literal becoming available for execution when it holds the tokens for all the shared variables that it uses. If a literal fails, backtracking to the literal which was the producer of a token allows the skipping of irrelevant speculative computation.

In the bit vector model, each variable in a clause is associated with a bit vector which has bits set for each literal in the clause which uses the variable. Each literal has a bit-vector mask associated with it, indicating the position of the literal in the clause. In the example shown in figure 2.4, the third clause has four variables, `A, B, DA` and `DB` with the bit vectors of 10 for `A` and `DA` and 01 for `B` and `DB`. The literal masks are 00 for `diff(A, DA)` and 10 for `diff(B, DB)`. If a variable is bound to a ground term, then the bit vector for the variable is set to all zeros. If two variables become dependent, then the bit vectors of both variables are or-ed together.

The finish vector is a bit vector where the bits are set to 0 as each literal completes. In the above example, the finish vector is 11 at the start of the clause, 01 if `diff(A, DA)` has completed and 00 when all literals have completed.

Detecting whether a literal G is ready to run consists of seeing whether $(\vee v_V) \wedge v_G \wedge F$ is zero for all variables V in G, where v_V is the bit vector for variable V, v_G is the literal mask and F is the finish vector.

In the example, if `diff/2` is called with `?-diff(x + c, D)`, then $v_A = 00, v_B = 00, v_{DA} = 10, v_{DB} = 01$, as `A` and `B` are both ground. For `diff(A, DA)` the readiness condition is $(00 \vee 10) \wedge 00 \wedge 11 = 00$. For `diff(B, DB)` the readiness condition is $(00 \vee 01) \wedge 10 \wedge 11 = 00$. As both readiness conditions are zero, both literals can be run in parallel.

If the call is `?-diff(X + X, D)` then $v_A = 11, v_B = 11, v_{DA} = 10, v_{DB} = 01$, as `A` and `B` are dependent on each other. For `diff(A, DA)` the readiness condition is $(11 \vee 10) \wedge 00 \wedge 11 = 00$. For `diff(B, DB)` the readiness condition is $(11 \vee 01) \wedge 10 \wedge 11 = 10$. The first literal is ready to run, the second literal must wait. After the first literal completes, the finish mask is set to 01 and the readiness condition for the second literal is now $(11 \vee 01) \wedge 10 \wedge 01 = 00$. The second literal can now be safely evaluated.

2.2.2 Static Detection of Independent And-Parallelism

Dynamic detection of IAP is expensive, especially the tests for groundness and variable independence, although groundness tests can be cached [DeG84]. An alternative to expensive run-time checks is to perform a static analysis of the data-flow dependencies of the program for some top-level goal and generate a single call-graph for the goal.

The method used in [Cha85] is to use a static analysis where variables are classified into sets of ground variables, independent variables and groups of variables that may be dependent on each other. The most pessimistic assumptions are made about variable aliasing, ensuring safe parallel execution.

2.2.3 Conditional Graph Expressions

The purely dynamic models of IAP tend to produce excess run-time testing. Static analysis restricts the amount of parallelism available. Hybrid methods, such as program graph expressions [DeG84] and conditional graph expressions (CGEs) [Her86a, Her86b] attempt to tread a path between the two extremes.

CGEs consist of compiled expressions specifying the conditions under which a set of literals or other CGEs can be run in parallel; these conditions can be evaluated at run-time. CGEs have the form `(`C `=>` G`)` where G is a list of literals and other CGEs and C is a list of conditions. The conditions can be any of the tests: `true`, `false`, `ground(`*`Vars`*`)` or `indep(`*`Vars`*`)`. The test `ground(`*`Vars`*`)` is true if all variables in $Vars$ are ground. The test `indep(`*`Vars`*`)` is true if all variables in $Vars$ are mutually independent.

If the conditions in a CGE all evaluate to true, then the list of literals or CGEs can be executed in parallel. Otherwise, the list must be executed sequentially.

```
p(a).
p(b).

q(c).
q(d).

r(X, Y) :- p(X), q(Y)

?-r(X, Y)
```

Figure 2.6: An Example Program Causing a Cross-Product

Applying the principles of CGEs to the example differentiator produces:

```
diff(A + B, DA + DB) :-
       (indep(A, B), indep(A, DB), indep(B, DA), indep(DA, DB) =>
              diff(A, DA)
              diff(B, DB)
       )
```

This CGE will detect most IAP, although it will miss some parallelism that the bit-vector method would catch. Eg. the parallelism in `?-diff(X + c, X + 0)` will be detected by the bit-vector method, but will be rejected by the `indep(X, X)` test.

Conditional graph expressions are well suited to optimisation by compile-time analysis, as the expressions can be grouped and manipulated by various forms of static analysis [MH90, XG88].

Conditional graph expressions also provide a means for handling nondeterministic and-parallelism. A failure while executing sequentially can be handled in the normal backtracking manner. A failure inside a CGE which is executing in parallel can cause all parallel calls to be killed and the computation to backtrack to the first choice outside the CGE. A failure outside a parallel CGE which backtracks into the CGE needs to search (right to left) along the list of goals in the CGE for a choice; the goals to the right of the choice then need to be restarted.

2.2.4 Combining Independent And-Parallelism and Or-Parallelism

Independent and-parallelism and or-parallelism are essentially orthogonal in their effects on a program. The main implementation difficulty that combining the two presents is the effect of two subgoals running in parallel producing multiple answers.

For example, in the program shown in figure 2.6 the subgoals `p(X)` and `q(Y)` can clearly be run in and-parallel. However, if or-parallelism is allowed in these subgoals then each subgoal can independently produce a set of or-parallel bindings, {{ X/a }, { X/b }} for `p(X)` and {{ Y/c }, { Y/d }} for `q(Y)`. These solutions need to be combined via some sort of cross-product operation:

$$\{\{\text{ X/a }\},\{\text{ X/b }\}\} \otimes \{\{\text{ Y/c }\},\{\text{ Y/d }\}\} =$$
$$\{\{\text{ X/a, Y/c }\},\{\text{ X/a, Y/d }\},\{\text{ X/b, Y/c }\},\{\text{ X/b, Y/d }\}\}$$

The And/Or process model [Con83], and its practical realisation in OPAL [Con92] uses a tree of and- and or-processes to collect solutions. Or-processes collect incremental copies of non-ground terms.

The PEPSys system [WR87] uses hash windows (section 2.1.1) to maintain or-parallelism. Creating a cross-product essentially means creating a cross-product of the candidate hash windows. IAP ensures that there will be no conflicting variable bindings in the hash windows created by different and-branches. Join cells are used to link hash windows for each possible element of the cross-product.

The ACE system [GH91] is a combination of conditional graph expressions and the copying model for or-parallelism. A group of workers executing a set of IAP subgoals makes a single area, which can be copied in total.

```
p(0, []).
p(N, [N | R]) :- N > 0, N1 is N - 1, p(N1, R).

sum([], S, S).
sum([N | R], S, S1) :- S2 is S + N, sum(S2, S1).

?-p(5, L), sum(L, 0, S).
```

Figure 2.7: An Example Dependent And-Parallel Program

The Reduce-OR model [Kal87] maintains sets of variable bindings, called tuples, for each branch of an and-parallel computation. These tuples are lazily combined where the execution graph joins to make a cross product.

The AO-WAM [GJ89] builds a tree of and- and or-nodes, extended by crossproduct- and sequential-nodes. The crossproduct-nodes combine solutions in a similar manner to the Reduce-OR model.

2.3 Dependent And-Parallelism

Dependent and-parallelism (DAP) or stream and-parallelism takes a view of parallelism similar to Hoare's communicating sequential processes [Hoa78]. Subgoals within a clause are executed as individual processes, with shared variables acting as conduits of information between the processes.

An example program capable of DAP is shown in figure 2.7. If called with a number and variable as arguments, the `p/2` predicate produces a stream of numbers, with the variable being progressively instantiated to form a list. The `sum/3` predicate can incrementally consume this list of numbers, constructing a partial sum as each number is produced by `p/2`. The shared variable `L` acts as a communication channel between the two subgoals, synchronising the two processes.

Clearly, this example is expected to act as a producer-consumer pair, with `p/2` acting as the producer and `sum/3` acting as the consumer. However, Prolog-like logic programming languages are inherently modeless, and conditionally bind variables while searching for a solution. If `sum/3` is called with an uninstantiated first argument, then it will try the first clause, conditionally binding the variable to `[]`. However, `p/2` is also executing at this time and will attempt to bind the variable to `[5 | L1]`. Some sort of mode information is needed to identify the expected producers and consumers of bindings.

If a producer of a binding makes a conditional binding, then this binding will be used by any consumer which shares a variable with the producer. If a failure occurs, then some form of distributed backtracking is needed, with consumers being resynchronised.

2.3.1 Committed Choice Languages

The distributed backtracking problem, described above, led to an abandonment of the standard Prolog-style nondeterminism (*don't know nondeterminism*) in exchange for a form of nondeterminism which ensures that there is only a single solution to a query, eliminating the problems of backtracking (*don't care nondeterminism*). If there are several solutions to a goal, then all solutions, bar one, are nondeterminately eliminated. The computation then commits to the remaining solution. This process of commitment gives the class of languages that support this feature the name of Committed Choice Languages (CCLs).

The various committed choice languages: Concurrent Prolog [Sha83], Parlog [CG86], GHC [Ued86] and KL1 [UC90] all share similar features. Over time, these languages have devolved as features that are difficult to implement and do not seem to be needed by programmers are stripped from them. An entertaining review of the CCLs and their devolution can be found in [Tic95].

CP [Sar87] is a formal unification of the various features of don't know and don't care nondeterminism, and the various synchronisation features that different CCLs supply. The Andorra model, discussed in the next chapter, and Parallel NU-Prolog [Nai88] have similar behaviour to CCLs, but allow restricted nondeterminism.

Syntax

Clauses in CCLs are written as

$$H \texttt{ :- } G_1, \ldots, G_n \mid B_1, \ldots, B_m$$

where H is the head of the clause, $G_1, \ldots, G_n$ is the *guard* and $B_1, \ldots, B_m$ is the *body*. The | element is the commit operator, separating the guard from the body. When a predicate is called, all clauses in the predicate attempt to solve their guards in parallel. If a guard succeeds, then any other non-failed guards are pruned, and the computation commits to that clause and begins executing the body atoms. The program in figure 2.7, rewritten in the CCL style would be:

```
p(0, []).
p(N, [N | R)) :- N > 0 | N1 is N - 1, p(N1, R).

sum([], S, S).
sum([N | R], S, S1) :- | S2 is S + N, sum(S2, S1).
```

Flat CCLs restrict guard atoms to being primitive operations, such as unification and arithmetic comparison, as opposed to deep guards, where the guards may be arbitrary literals. Examples of flat CCLs are FCP [YKS90] and Flat GHC [UF88]. The main motivation for introducing flat languages is the difficulty of implementing deep guards. A full implementation of deep guards requires separate binding environments for each guard computation, making it as least a hard a problem as or-parallelism. Crammond's JAM [Cra88] for Parlog only allows one deep guard to be evaluated at a time, allowing deep guards, but eliminating a source of parallelism.

Modes

CCLs also need to provide some mechanism for specifying which subgoals are producers of bindings and which are consumers — modes. Each CCL provides different means of supplying mode information. The different ways of declaring modes, roughly in decreasing order of flexibility (and implementation difficulty) are:

1. **Read Only Variables** Concurrent Prolog provides read-only variable annotations. Variables that are marked with a `?` in a literal are read-only and may not be bound by that literal. In the example in figure 2.7, the initial query would be written as `?-p(5, L), sum(?L, 0, S)`.

2. **Ask:Tell** Clauses in Concurrent Prolog may have an Ask:Tell part at the start of the clause. Constraints in the Ask part of the clause must be supplied externally to the clause. The Tell part of the clause atomically exports the bindings that it contains. The first clause of `p/2` in the example would be written as `p(N, L) :- N = 0 : L = []`.

3. **Suspension** The GHC and KL1 suspension rule forces a clause to suspend when a guard attempts to bind a variable that is external to the clause. The first clause of `p/2` in the example would be written as `p(N, L) :- N = 0 | L = []`. Suspension is similar to the Ask:Tell notation above, but the body part is not guaranteed to be atomic.

4. **Mode Declarations** Parlog and Parallel NU-Prolog both use mode declarations on predicates to indicate which arguments are input and which are output. Arguments which are marked as input cause the goal to suspend until the argument is sufficiently instantiated to satisfy any candidate guards without requiring further variable bindings. if the guard attempts to bind the argument. In the above example, `sum/3` has a mode of `?-mode sum(?, ?, ↑)` in Parlog and `?-lazyDet sum(i, i, o)` in Parallel NU-Prolog and calls to `sum/3` would suspend until the first argument is bound, although the argument need not be ground. Mode declarations are less flexible than rules based on individual variables. The program below is an example of a GHC program which can not be given a simple mode declaration:

```
and(X, Y) :- X = 0 | Y = 1.
and(X, Y) :- X = 1 | Y = 0.
and(X, Y) :- Y = 0 | X = 1.
and(X, Y) :- Y = 1 | X = 0.
```

2.3.2 Reactive Programming Techniques

Dependent And-Parallelism allows an array of programming techniques, that are impossible in ordinary Prolog-like systems, with their left-to-right computation rule. Since goals can be suspended until information becomes available, networks of processes can be created, passing streams of data between themselves.

Stream Programming

Lists in DAP can be regarded as streams of data, with producers and consumers acting as processes passing streams of messages to each other. As an example, the following predicate (in GHC) filters an incoming stream, removing any adjacent duplicate elements:

```
unique([], O) :- true | O = [].
unique(I, O) :- I = [_] | O = I).
unique([X, X | I1], O) :- true | unique([X | I1], O).
unique([X, Y | I1], O) :- X ~= Y | O = [X | O1], unique([Y | O1], O).
```

Duplicating streams is a matter of repeating variables in a goal. For example, `unique(I, U), replace(U, a, b, U1), replace(U, a, c, U2)` has two `replace/4` filters, each being fed from the same stream contained in `U`.

The `merge/3` predicate can be used to combine two streams into a single stream:

```
merge([], I2, O) :- true | O = I2.
merge(I1, [], O) :- true | O = I1.
merge([X | I1], I2, O) :- true | O = [X | O1], merge(I1, I2, O1).
merge(I1, [X | I2], O) :- true | O = [X | O1], merge(I1, I2, O1).
```

This predicate relies on the commit operator eliminating alternate clauses when a guard has been satisfied. When a binding appears on an input stream, an eligible clause is committed to, regardless of the state of the other input stream. The merge predicate produces a nondeterminate merging of the two streams, with the order of the output stream matching the order that elements appeared on the two input streams.

Object Oriented Programming

Objects can be represented as processes which communicate using streams of messages. A predicate receives the messages and responds to each message appropriately; the clauses of the predicate provide the method definitions for the object. An example of an object implementation is:

```
io([], _) :- true | true.
io([open(Name) | R], _) :- true |
       open_file(Name, Handle),
       io(R, Handle).
io([close | R], Handle) :- true |
       close_file(Handle),
       io(R, x).
io([write(C) | R], Handle) :- true|
       write_file(Handle, C),
       io(R, Handle).
```

This object represents a simple file stream, which receives an `open` message, followed by a sequence of `write` messages, followed by a `close` message. Access to the object is granted by the message stream. If there are to be several objects that use this object, then each object produces a message stream and the message streams are merged. An example to the `io/2` predicate in use is

```
?-io([open(foo)|Io],x), merge(Io1, Io2, Io), writer1(Io1), writer2(Io2).
```

In this example, the object is represented by the stream on `Io`. The two writers produce streams of messages which are merged and forwarded to the `Io` stream.

Incomplete Messages

Incomplete messages [Sha86] extend the object-oriented model described above by providing a mechanism for back communication. If an uninstantiated variable is included in the arguments of a message, that variable may be bound by the predicate which is handling the object's messages. An example of incomplete messages is this stack implementation:

```
stack([], _) :- true | true.
stack([push(X) | R], S) :- true | stack(R, [X | S]).
stack([pop(X) | R], S) :- true | S = [X | S1], stack(R, S1).
stack([top(X) | R], S) :- true | S = [X | _], stack(R, S).
```

In this example, if the `pop` message is sent with an uninstantiated variable as its argument, then the variable will be bound to whatever is on top of the stack.

2.3.3 Don't Know Nondeterminism and Dependent And-Parallelism

The CCLs described above all rely on don't care nondeterminism to avoid the sticky problems of distributed backtracking. Other approaches combine don't know nondeterminism and DAP.

Ptah [Som87, SRV88, Som89] uses strict mode declarations to identify the producers and consumers of variable bindings. The strict mode declarations allow a data-flow graph to be built for the computation. If a part of the computation fails, the source of the original binding that caused the failure is known and which parts of the computation must be retried and which parts need to be restarted. can be deduced.

Ptah allows the reactive programming of section 2.3.2. However, the amount of mode information needed to identify producers and consumers can make for a quite onerous task, removing the attractive conciseness of logic programming.

Shen's Dynamic Dependent And-Parallel Scheme (DDAS) [She92, She93] provides transparent exploitation of and-parallelism. Conceptually, the scheme is a token-passing system similar to the IAP model discussed in section 2.2.1. Each variable has a producer token which is initially given to the left-most and-node that refers to the variable. As and-nodes complete, producer tokens are passed on to the next and-node that refers to the variable. If an and-node which does not hold the producer token for a variable attempts to bind the variable, it suspends until an and-node to the left binds the variable, or it acquires the producer token.

In practise, the DDAS is implemented by using a variety of the CGEs discussed in section 2.2.3. CGEs are used to partition goals into independent groups of goals, with the goals within the groups potentially dependent on each other. Rather than assign producer tokens to each variable, each group has a single producer token that passes from left to right along the group.

The DDAS can be regarded as an attractive form of IAP; it transparently provides the same behaviour as Prolog, without some of the restrictions of IAP. The reproduction of Prolog-like behaviour means that the reactive programming techniques discussed in section 2.3.2 are not possible, although allowing a flexible computation rule for the DDAS is an intriguing idea.

2.4 Other Forms of Parallelism

The preceding sections have discussed the major forms of parallelism inherent in logic programs. These forms of parallelism are those considered throughout the rest of this thesis. However, there are a number

of additional approaches to parallelism in logic programs; a brief summary of these approaches is given below.

Both or- and independent and- parallelism attempt to transparently extract parallelism from Prolog-like programs. Dependent and-parallelism, despite the use of CCLs, still attempts to supply an implied model of parallelism. Process-oriented logic programming languages, such as Delta-Prolog [PN84] or CS-Prolog [FF92] use explicit message passing operators to transmit and receive messages between essentially unconnected Prolog processes.

Data-flow models, such as Kacsuk's 3DPAM [Kac92] or Zhang's DIALOG [ZT91], model the and-or tree by means of tokens passing between the nodes of the tree.

Reform parallelism [Mil91] is a form of vector parallelism where recursively defined predicates are flattened into iterative loops and constructed so as to allow execution on a vector parallel processor.

Chapter 3

The Andorra Model

The basis of the Andorra model can be reduced to a single statement: *"Do the determinate bits first."* This simple statement provides both a way of unifying dependent and- and or-parallelism and an efficient computation rule for logic programs. The Andorra Kernel Language allows nondeterministic independent and-parallelism to be also united under the Andorra flag.

Dependent and-parallelism is much simpler to implement when the computation is determinate. The problems of distributed backtracking over several cooperating computations tends to prevent mixing dependent and- and or-parallelism, with the exceptions of DDAS and Ptah. Independent and-parallelism avoids the major problems of distributed backtracking by prohibiting and-parallel calls from influencing each other. As a result, dependent and-parallel languages tend to be committed choice languages — Concurrent Prolog, Parlog, GHC — which enforce determinism.

The roots of the Andorra model can be found in Naish's thesis [Nai86]. Naish proposed that a desirable computation rule would choose atoms in the following order: tests that were likely to fail, deterministic calls, non-deterministic calls with a finite number of solutions, non-deterministic calls likely to cause loops and uninstantiated system predicates (eg. negation). The Andorra model collapses this list into a simple distinction between deterministic calls and non-deterministic calls. This distinction can be made by simple run-time tests, making the Andorra model an efficient computation rule [Nai93].

Early versions of the Andorra model for dependent and-parallelism go back to Yang's P-Prolog [YA87], where sets of alternate clauses were chosen by means of explicitly grouping them together. Naish's parallel NU-Prolog [Nai88] is also implicitly organised about the Andorra model; goals delay until sufficient information becomes available to commit to a single clause.

3.1 The Basic Andorra Model

The Andorra model was first named by D.H.D. Warren at a Gigalips meeting in 1987, who pointed out that determinism could be made the basis of transparently exploiting dependent and-parallelism. This model is the *Basic Andorra Model* or *BAM*. A description of the BAM can be found in Santos Costa's thesis [SC93]. The BAM recognises two basic operations:

- Any literals that can be detected as deterministic are reduced (in parallel, if possible)

$$(A_1, \ldots, A_i, \ldots, A_n) \Rightarrow (A_1, \ldots, B_1, \ldots, B_m, \ldots, A_n)$$

- If no literals are detected to be deterministic, then a goal is selected, and forked into a set of alternate configurations.

$$(A_1, A_2, \ldots, A_n) \Rightarrow (B_{11}, \ldots, B_{1m_1}, \ldots A_2, \ldots, A_n) \vee \cdots \vee (B_{l1}, \ldots, B_{lm_l}, \ldots A_2, \ldots, A_n)$$

As an example of the BAM, consider the program

```
p(b, a, a).
p(a, b, a).
p(a, a, b).
```

and the query `?-p(X, Y, Z), p(Z, W, W)`. Initially, `p(X, Y, Z)` could match any of the three clauses of `p/3`. However, this query can be computed determinately, since only one clause of `p/3` matches `p(Z, W, W)`. The first step of a BAM computation, therefore, is the reduction step `p(X, Y, Z), p(Z, W, W)` $\Rightarrow$ `p(X, Y, b)`. The goal is now `p(Z, Y, b)` which now also matches a single clause of `p/3`, and can also be reduced with a final substitution of { W/a, X/a, Y/a, Z/b }.

The BAM is an idealised description of the Andorra model. To become a practical system, an instance of the BAM needs to supply such details as how determinism is detected and how extra-logical features (eg. cut) are handled. There are a number of applications of the BAM: Andorra-I [SCWY91b] is a parallel version of the BAM which executes Prolog programs. Andorra-I includes a sophisticated pre-processor that allows Andorra-I programs to act exactly like a Prolog program. Andorra Prolog [HB88] is an initial attempt to apply the Andorra model to Prolog. Pandora [BG89] uses the Andorra model in conjunction with Parlog. NUA-Prolog [PN91] is a basic application of the BAM to Prolog, using negations instead of cuts.

3.2 Andorra Prolog

Andorra Prolog [HB88] is an instance of the BAM. Andorra Prolog provides semantics for cut and commit operators, missing from the BAM, and formalises the execution model in terms of a series of configurations. While never fully implemented, unlike Andorra-I, Andorra Prolog is of interest as one of the predecessors of the AKL, discussed in section 3.4. In particular, the configuration-based approach forms a natural bridge between Andorra Prolog and the AKL.

3.2.1 Execution Model

The execution model presented here is based on that of Haridi and Brand [HB88]. The implicit node-tree built in [HB88] has been made explicit; the explicit node-tree makes the relationship between Andorra Prolog and the AKL (section 3.4) more apparent.

Programs in Andorra Prolog consist of a set of definite clauses in the form: H `:-` G, B The head, H, is a single atom. The guard, G, and the body, B, are sequences of atoms, with G restricted to simple tests, such as `==/2`, `</2` or `atom/1`.

Given a substitution θ, an atom A, and a clause $S \equiv H$ `:-` G, B, S is a *candidate clause* for A if $A\theta$ unifies with H, and G is satisfiable in the context of $\theta\sigma$, where $\sigma = \text{mgu}(A\theta, H)$. In an Andorra Prolog computation, each atom in a goal is associated with a list of candidate clauses.

A *goal* is a pair (A, C), where A is an atom and $C = [C_1, \ldots, C_n]$ is a list of candidate clauses for A. (A, C) is *determinate* if C contains a single clause. A *configuration* is $(L, \theta, N)_{mode}$ where L is a list of goals, θ is a substitution, N is a list of child configurations and $mode$ is one of And, Or or $Failure$. An initial query, `?-` $A_1, \ldots, A_n$, is written as $([(A_1, C_1), \ldots, (A_n, C_n)], \epsilon, [])_{And}$, where each C_i is the set of candidate clauses for A_i and ϵ is the empty substitution.

An Andorra Prolog computation then proceeds using the operations of failure, and-reduction, and-extension and or-extension, in the following order of priority:

1. **Failure:** If the configuration is $(L, \theta, N)_{And}$ and there is a goal in L with an empty clause list, then the configuration is changed to $(L, \theta, N)_{Failure}$.

2. **And-reduction:** If the configuration is $(L, \theta, N)_{And}$ and there is a determinate goal, $L_i = (A, [H$ `:-` $G, B])$ in L, then $A\theta$ is unified with H to give a new substitution σ. The configuration is then changed to

$$\left(\left[\begin{array}{c} L'_1, \ldots, L'_{i-1}, \\ (G_1, C_{G_1}), \ldots, (G_l, C_{G_l}), (B_1, C_{B_1}), \ldots, (B_m, C_{B_m}), \\ L'_{i+1}, \ldots, L'_n \end{array} \right], \theta\sigma, N \right)_{And}$$

```
n(0, 0).                                          (C1)
n(N, s(R)) :- N > 0, N1 is N - 1, n(N1, R).       (C2)

e(0).                                             (C3)
e(s(s(E))) :- e(E).                               (C4)

?-n(3, V), e(V).
```

$$([(\texttt{n(3, V)}, [C2]), (\texttt{e(V)}, [C3, C4])], \epsilon, [])_{And}$$
$$\Downarrow$$
$$([(\texttt{n(2, R)}, [C2]), (\texttt{e(V)}, [C4])], \{\ \text{V/s(R)}\ \}, [])_{And}$$
$$\Downarrow$$
$$([(\texttt{n(1, R1)}, [C2]), (\texttt{e(R1)}, [C3, C4])], \{\ \text{V/s(R), R/s(R1)}\ \}, [])_{And}$$
$$\Downarrow$$
$$([(\texttt{n(0, R2)}, [C1]), (\texttt{e(R1)}, [C4])], \{\ \text{V/s(R), R/s(R1), R1/s(R2)}\ \}, [])_{And}$$
$$\Downarrow$$
$$([(\texttt{e(R1)}, [])], \{\ \text{V/s(R), R/s(R1), R1/s(R2), R2/0}\ \}, [])_{And}$$
$$\Downarrow$$
$$([(\texttt{e(R1)}, [])], \{\ \text{V/s(R), R/s(R1), R1/s(R2), R2/0}\ \}, [])_{Failure}$$

Figure 3.1: An Example And-Parallel Andorra Prolog Execution

where $G = G_1, \ldots, G_l$, $B = B_1, \ldots, B_m$, C_A is the candidate clause list for atom A and L'_j is L_j with all clauses in the candidate clause list for L_j which are not compatible with σ removed.

The guard part of the clause needs to be included in the final configuration, since it may include such atoms as $X < 1$, where X is a variable.

3. **And-extension:** If the configuration is $(L, \theta, N)_{And}$ and there are no determinate goals in L then the configuration is changed to $(L, \theta, N)_{Or}$.

4. **Or-extension:** If the configuration is $([(A, C), L_2, \ldots, L_n], \theta, N)_{Or}$ and $C = [C_1, \ldots, C_m]$ is non-empty then the configuration is changed to:

$$([(A, [C_2, \ldots, C_m]), L_2, \ldots, L_n], \theta, N \cdot [([(A, [C_1]), L_2, \ldots, L_n], \theta, [])_{And}])_{Or}$$

where $N \cdot M$ is the concatenation of two lists.

The Andorra Prolog execution model builds a tree of nodes, with and- and failure-nodes at the leaves and or-nodes at higher levels of the tree. Since or-extension chooses the left-most and first clause for extension, the Andorra Prolog search rule closely follows the Prolog search rule.

Andorra Prolog computations are implicitly parallel. And-parallelism occurs when several and-reductions are applied to a single configuration concurrently. Or-parallelism occurs if separate configurations are reduced or extended concurrently. Example Andorra Prolog computations for and-parallelism and or-parallelism are shown in figures 3.1 and 3.2 respectively.

3.2.2 Commit

Commits are permitted in Andorra Prolog immediately after a guard; a clause can be written as H `:-` G, `|` , B. Tests in the guard, G, must be completely solved before the commit operator is applied, although unifications may proceed. When the guard is completely solved, the commit operator prunes all other candidate clauses from the list of candidate clauses. If several candidate clauses have solved guards, then a single clause is (nondeterministically) chosen. An example of the commit operator is shown in figure 3.3.

At or-extension, unsolved guards and their commit operators are carried with the or-extended configurations. When the guard is solved, the commit operator eliminates all other nodes in the parent or-configuration. See figure 3.4 for an example of commit occurring after or-extension.

```
nondet(a).    (C1)
nondet(b).    (C2)

?-nondet(X).
```

$$([(\texttt{nondet(X)}, [C1, C2])], \epsilon, [])_{And}$$
$$\Downarrow$$
$$([(\texttt{nondet(X)}, [C1, C2])], \epsilon, [])_{Or}$$
$$\Downarrow$$
$$([(\texttt{nondet(X)}, [C2])], \epsilon, [([(\texttt{nondet(X)}, [C1])], \epsilon, [])_{And}])_{Or}$$
$$\Downarrow$$
$$([], \epsilon, [([], \{\ \text{X/a}\ \}, [])_{And}, ([(\texttt{nondet(X)}, [C2])], \epsilon, [])_{And},])_{Or}$$
$$\Downarrow$$
$$([], \epsilon, [([], \{\ \text{X/a}\ \}, [])_{And}, ([], \{\ \text{X/b}\ \}, [])_{And}])_{Or}$$

Figure 3.2: An Example Or-Parallel Andorra Prolog Execution

```
max(X, Y, X)  :-  X >= Y,  | .  C1
max(X, Y, Y)  :-  X =< Y,  | .  C2

?-max(5, 5, Z).
```

$$([(\texttt{max(5, 5, X)}, [C1, C2])], \epsilon, [])_{And}$$
$$\Downarrow \text{ Both } \texttt{5 >= 5} \text{ and } \texttt{5 =< 5} \text{ are solved.}$$
$$([], \{\ \text{Z/5}\ \}, [])_{And}$$

Figure 3.3: Example of a Andorra Prolog Commit

```
p(X, a)  :-  X >= 0,  | .  C1
p(X, b)  :-  X =< 0,  | .  C2

q(0).                    C3
q(2).                    C4

?-p(Y, Z), q(Y).
```

$$([(\texttt{p(Y, Z)}, [C1, C2]), (\texttt{q(Y)}, [C3, C4])], \epsilon, [])_{And}$$
$$\Downarrow$$
$$([(\texttt{p(Y, Z)}, [C1, C2]), (\texttt{q(Y)}, [C3, C4])], \epsilon, [])_{Or}$$
$$\Downarrow$$
$$\left([], \epsilon, \left[\begin{array}{c} ([\{Y >= 0,\ \texttt{|}\ \}, (\texttt{q(Y)}, [C3, C4])], \{\ \text{Z/a}\ \}, [])_{And}, \\ ([\{Y =< 0,\ \texttt{|}\ \}, (\texttt{q(Y)}, [C3, C4])], \{\ \text{Z/b}\ \}, [])_{And} \end{array}\right]\right)_{Or}$$
$$\Downarrow$$
$$\left([], \epsilon, \left[\begin{array}{c} ([], \{\ \text{Z/a}\ \}, [([\{Y >= 0,\ \texttt{|}\ \}], \{\ \text{Y/0}\ \}, [])_{And}, ([\{Y >= 0,\ \texttt{|}\ \}], \{\ \text{Y/2}\ \}, [])_{And}])_{Or}, \\ ([], \{\ \text{Z/b}\ \}, [([\{Y =< 0,\ \texttt{|}\ \}], \{\ \text{Y/0}\ \}, [])_{And}, ([\{Y =< 0,\ \texttt{|}\ \}], \{\ \text{Y/2}\ \}, [])_{Failure}])_{Or} \end{array}\right]\right)_{Or}$$
$$\Downarrow \text{ Choose one of the possible commits}$$
$$([], \epsilon, [([], \{\ \text{Z/a}\ \}, [])_{Or}, ([], \{\ \text{Z/b}\ \}, [([], \{\ \text{Y/0}\ \}, [])_{And},])_{Or}])_{Or}$$

Figure 3.4: Example of a Andorra Prolog Commit after Or-Extension

The commit operator, as defined for Andorra Prolog, does not exactly act in an intuitive manner. For example, the program

```
p(a)  :-   | .
p(b)  :-   | .
```

works in a similar manner to the GHC program,

```
p(X)  :-  true,  | , X = a.
p(X)  :-  true,  | , X = b.
```

rather than the expected[1] GHC program,

```
p(X)  :-  X = a,  | , true.
p(X)  :-  X = b,  | , true.
```

Explicit test predicates are needed to make the Andorra Prolog program act like a GHC program. The program above can be rewritten as

```
p(X)  :-  X == a,  | .
p(X)  :-  X == b,  | .
```

The `==/2` predicate only accepts input bindings. The predicate is therefore forced to wait until `X` is bound.

3.2.3 Cut

The cut operator (!) is similar to the Prolog cut operator. The use of cut in Prolog assumes a left-to-right computation rule. For example, given the program:

```
p(1) :- !.
p(2).
```

the query `?-p(X), X = 2` would fail in Prolog, but succeed in an Andorra Prolog computation. This problem occurs whenever cuts which remove solutions from the program occur — red cuts, as opposed to green cuts which simply eliminate redundant solutions.

The solution suggested in Haridi and Brand's description of Andorra Prolog is to have a relaxed form of cut, one which acts as a normal cut during or-extension, but relaxes and acts in a similar manner to a commit during and-reduction.

The implementation of Andorra-I [SCWY91b] provides a mechanism that enforces the semantics of the Prolog cut. A preprocessor identifies red cuts, and inserts a sequencing operator (written as `::`) into the code. All goals to the left of the sequencing operator must complete before any goals to the right are executed. In the above example, the query would be re-written as `?-p(X) :: X = 2`, producing Prolog-like behaviour.

[1] Expected in the sense that unifications to the left of the commit operator are intuitively part of the guard.

3.3 The Extended Andorra Model

The BAM described in section 3.1 is simple, but it cannot detect deep determinism. For example, it would be useful to extract determinism from a program such as:

```
p(X, Y, a) :- member(X, Y).
p(X, Y, b) :- not member(X, Y).
```

In this case, a sufficiently instantiated call to `p/3` would be deterministic. However the BAM execution model would not recognise this possibility[2], and `p/3` would have to wait for forking. Generally, implementations of the BAM are flat, relying on head arguments and a few test predicates to detect determinism.

The BAM also prevents nondeterministic independent and-parallelism, as a single goal is chosen for forking.

The Extended Andorra Model (EAM) was designed by Warren to allow the detection of most parallelism within a logic program. A Description of the EAM can be found in Santos Costa's thesis [SC93]. EAM computations are formally defined in terms a set of rewrite rules on trees of *boxes:*.

And-box: $[\exists X : \theta \wedge C_1 \wedge \cdots \wedge C_m]$ where each X is the set of variables local to the and-box, θ is a set of constraints on variables external to the and-box and each C_i is either an atom or box.

Or-box: $\{C_1 \vee \cdots \vee C_n\}$ where each C_i is a box.

Choice-box: $\{G_1 \ \% \ C_1 \vee \cdots \vee G_n \ \% \ C_n\}$ where $\%$ is a guard operator (either cut or commit), each G_i is a list of guard atoms and boxes and each C_i is a list of body atoms and boxes.

The EAM computation proceeds by applying the following rewrite rules:

Reduction:

$$A \Rightarrow \{[\exists Y_1 : \theta_1 \ \% \ C_i] \vee \cdots [\exists Y_n : \theta_n \ \% \ C_n]\}$$

Reduction expands an atom into an or-box.

Promotion:

$$[\exists X : S \wedge [\exists Y, W : Z = \theta(W) \wedge C] \wedge T] \Rightarrow [\exists X, W : X = \theta(W) \wedge S \wedge [\exists Y : C] \wedge T]$$

Promotion moves constraints from an inner (determinate) and-box to an outer and-box.

Substitution:

$$[\exists X, Z : Z = \theta(W) \wedge C] \Rightarrow [\exists X : C\theta]$$

Substitution imposes a constraint (substitution) on a variable and propagates the consequences of the substitution throughout the tree.

Forking:

$$[\exists X : \{C_1 \vee \cdots \vee C_n\} \wedge G] \Rightarrow \{[\exists X : C_1 \wedge G] \vee \cdots \vee [\exists X : C_n \wedge G]\}$$

Forking distributes a guard across a conjunction. Forking usually implies that the configuration inside the or-box has suspended on some variable. Forking is normally immediately followed by a promotion.

[2] Unless there was an *extremely* sophisticated compiler available.

The EAM prevents over-eager computation by two control rules. An and-box suspends if it attempts to constrain a variable external to it. Suspension of an and-box allows dependent and-parallelism. Forking is always tried last, allowing the Andorra principle to be applied.

If the producer of a variable binding can be identified, then the computation can be allowed to eagerly fork, with a system of *lazy copying* only copying the configuration of a forking operation when a forked box suspends. The forking rule then becomes:

$$[\exists X : \{C_1 \vee \cdots \vee C_n\} \wedge G] \Rightarrow \{[\exists X : C_1 \wedge G] \vee [\exists X : C_2 \wedge G_p] \vee \cdots \vee [\exists X : C_n \wedge G_p]\}$$

where the G_p are references to the original G.

To allow lazy copying, variables need to be classified as either *guessable*, *non-guessable* or other. Variables are marked as guessable if it is certain that the variable will be constrained nondeterministically. Variables are marked as non-guessable if forking should only be tried as a last resort. Variables can either be marked by some programmer supplied annotation, or by some pre-processor.

3.4 Andorra Kernel Language

The EAM represents an idealised model of parallelism in logic programming. The EAM is designed to extract the maximum amount of parallelism from logic programs without much attention being applied to efficiency or control. An actual implementation of the EAM needs to refine parts of the raw EAM execution model and provide some means for controlling execution.

The Andorra Kernel Langauge (AKL) is an instance of the EAM which uses guards to provide a simpler control model. Guarded clauses are used to separate guards and bodies, to allow deep guards and to identify which variables can be constrained. A number of different guard operators are permitted, describing a variety of pruning behaviours. The odd behaviour of the Andorra Prolog pruning operators (section 3.2.2) has been replaced by a simple set of well-behaved pruning rules.

The EAM suspends an and-box when a external variable is about to be constrained. Since the AKL uses guard operators to separate the speculative parts of a clause from the body of the clause, the AKL can speculatively pre-compute parts of a clause which would be suspended in a pure EAM computation.

The model for AKL presented here is a combination of the original Kernel Andorra Prolog (KAP) described by Haridi and Janson [HJ90] and the later Andorra Kernel Language [JH91]. The model retains the constraint-based description of the original KAP, but includes such elements as the `bagof/3` predicate introduced by the AKL. The AKL terminology of using `?` for wait operators, `|` for commit operators and `->` for conditional operators, instead of `:,` `|` and `!` respectively, has been used.

3.4.1 AKL Programs

Guarded clauses are built using the following grammar (seq. is an abbreviation of sequence)

$<$ *guarded clause* $>$ ::= $<$ *head* $>$ `:-` $<$ *guard* $><$ *guard operator* $><$ *body* $>$
$<$ *head* $>$ ::= $<$ *program atom* $>$
$<$ *guard* $>$ ::= $<$ *seq. of atoms* $>$
$<$ *body* $>$ ::= $<$ *seq. of atoms* $>$
$<$ *atom* $>$ ::= $<$ *program atom* $>|<$ *constraint atom* $>|<$ *aggregate* $>$
$<$ *aggregate* $>$ ::= `bagof`$(<$ *variable* $>, <$ *body* $>, <$ *variable* $>$
$<$ *guard operator* $>$ ::= `?` | `->` | `|`

Constraint atoms are formulas in some constraint system. The choice of the constraint system used is made by the implementation, but it at least needs to be capable of using the Prolog constraint system of syntactic equality on terms. The existence of some constraint algorithm is assumed (unification in the case of equality) that can establish the consistency of conjunctions of constraints, and simplify them as required.

Program atoms are atomic formulas of the form $p(t_1, \ldots, t_n)$ where $t_1, \ldots, t_n$ are terms. In a guard or body, a formula of the form $p(t_1, \ldots, t_n)$ can be treated as being equivalent to $X_1 = t_1, \ldots, X_n = t_n, p(X_1, \ldots, X_n)$, where $X_1, \ldots, X_n$ are distinct variables. The head atom can be rewritten as $p(X_1, \ldots, X_n)$ `:-` $X_1 = t_1, \ldots, X_n = t_n$.

A *predicate definition* is a finite sequence of guarded clauses, each with the same head atom and guard operator. A *program* is a finite sequence of predicate definitions.

The `bagof/3` predicate only allows variables as first and third arguments. More complex forms of `bagof/3` are possible, where terms are used instead of variables in these arguments. However, the more complex forms of `bagof/3` can be reduced to the simple form by rewriting **bagof** $(T_1; G; T_2)$ as **bagof** $(V; (G, V = T_1); U), U = T_2$.

3.4.2 Execution Model

The AKL execution model is described in terms of *configurations*, nested expressions built from atoms, and terms called boxes. Configurations can be built from the following grammar:

$$
\begin{array}{rcl}
< \textit{configuration} > & ::= & < \textit{and-box} > | < \textit{or-box} > \\
< \textit{and-box} > & ::= & \textbf{and}\,(< \textit{seq. of local goals} >; < \textit{constraint} >)_{< \textit{set of vars} >} \\
< \textit{or-box} > & ::= & \textbf{or}\,(\ < \textit{seq. of configurations} >\ ) \mid \textbf{fail} \\
< \textit{local goal} > & ::= & < \textit{atom} > | < \textit{choice-box} > | < \textit{bagof-box} > \\
< \textit{choice-box} > & ::= & \textbf{choice}\,(\ < \textit{seq. of guarded goals} >\ ) \mid \textbf{fail} \\
< \textit{bagof-box} > & ::= & \textbf{bagof}\,(< \textit{variable} >; < \textit{goal} >; < \textit{variable} >) \\
< \textit{guarded goal} > & ::= & < \textit{configuration} >< \textit{guard operator} >< \textit{seq. of atoms} > \\
< \textit{goal} > & ::= & < \textit{configuration} > | < \textit{local goal} > | < \textit{guarded goal} >
\end{array}
$$

In the following rules, R and S are used to denote sequences of configurations or goals, θ and σ are used to denote constraints, $\%$ to represent a generic guard operator, G to represent guard sequences of goals, B to represent body sequences of goals and V, W, X and Y to represent sets of variables.

In an and-box **and** $(R; \theta)_V$ the constraint, θ, is *quiet* if it only constrains variables occurring in V.

The execution model starts with a query, G, being written as **and** $(G; \text{true})_{vars(G)}$ and proceeds according to the following set of transition rules:

The *constraint imposition rule*

$$\textbf{and}\,(R, op(\sigma), S; \theta)_V \ \Rightarrow\ \textbf{and}\,(R, S; \sigma \wedge \theta)_V$$

is applied whenever $op(\sigma)$ is a constraint operation producing the constraint σ, and σ and the environment of the box are compatible.

The *environment synchronisation rule*

$$\textbf{and}\,(R, op(\sigma), S; \theta)_V \ \Rightarrow\ \textbf{fail}$$

occurs whenever σ and the environment of the box are incompatible.

The *local forking rule*

$$A \ \Rightarrow\ \textbf{choice}\,(\ \textbf{and}\,(G_1; \text{true})_{V_1}\ \%\ B_1, \ldots, \textbf{and}\,(G_n; \text{true})_{V_n}\ \%\ B_n\ )$$

converts a program atom A which does not occur in the body of a guarded goal into a choice-box. A `:-` $G_1\ \%\ B_1, \ldots, A$ `:-` $G_n\ \%\ B_n$ is the predicate definition of A, with the arguments of A substituted for formal parameters and the local variables of the ith clause replaced by the variables in V_i. The ordering of clauses within the predicate definition is important and must be preserved. The local forking rule is similar to a call in an ordinary Prolog execution.

The *failure propagation rule*

$$\textbf{and}\,(R, \textbf{choice}\,(), S; \theta)_V \ \Rightarrow\ \textbf{fail}$$

may be used to fail an and-box whenever one of the child choice-boxes has failed.

The *cut rule*

$$\textbf{choice}\,(\ R, \textbf{and}\,(; \sigma)_V\ \texttt{->}\ B, S\ )\ \Rightarrow\ \textbf{choice}\,(\ R, \textbf{and}\,(; \sigma)_V\ \texttt{->}\ B\ )$$

and the *commit rule*

$$\textbf{choice}\,(\ R, \textbf{and}\,(; \sigma)_V\ \texttt{|}\ B, S\ )\ \Rightarrow\ \textbf{choice}\,(\ \textbf{and}\,(; \sigma)_V\ \texttt{|}\ B\ )$$

are the pruning rules of AKL and are only applied if σ is quiet. The cut rule provides behaviour similar to the Prolog cut (`!`), and eliminates solutions after its application. The commit rule provides symmetrical pruning behaviour similar to the commit of GHC.

The above rules provide a mechanism for building a configuration corresponding to the guards which match some query. The Andorra model specifies that any guard which is now determinate can be promoted, and any constraints applied to the immediate environment. The *determinate promotion rule*

$$\textbf{and}\left(R, \textbf{choice}\left(\ \textbf{and}\left(;\theta\right)_V\ \%\ B\ \right), S;\sigma\right)_W \Rightarrow \textbf{and}\left(R, B, S;\sigma\wedge\theta\right)_{V\cup W}$$

is the Andorra part of AKL. If the guard operator is a conditional (`->`) or commit (`|`) guard, then θ must be quiet for the determinate promotion rule to take effect.

In addition to the determinate promotion rule, some mechanism is need to allow the controlled execution of non-determinate branches of the computation, after all other determinate avenues of computation have been explored. The *non-determinate promotion rule*

$$\begin{aligned}&\textbf{and}\left(R_1, \textbf{choice}\left(\ S_1, \textbf{and}\left(;\sigma\right)_V\ \texttt{?}\quad B, S_2\ \right), R_2;\theta\right)_W \Rightarrow\\ \textbf{or}\left(\ \textbf{and}\left(R_1, B, R_2;\theta\wedge\sigma\right)_{V\cup W}, \textbf{and}\left(R_1, \textbf{choice}\left(\ S_1, S_2\ \right), R_2;\theta\right)_W\ \right)&\end{aligned}$$

is used to provide controlled don't-know non-determinism. The non-determinate promotion rule is only applied to an and-box that is *stable*. An and-box is stable if no other rule can be applied to the and-box, and no external constraint can affect the and-box.

The *guard distribution rule*

$$\textbf{choice}\left(\ R, \textbf{or}\left(\ G, S\ \right)\ \%\ B, T\ \right) \Rightarrow \textbf{choice}\left(\ R, G\ \%\ B, \textbf{or}\left(\ S\ \right)\ \%\ B, T\ \right)$$

is applied to distribute the effects of a non-determinate promotion up to the next choice-box.

Bagof-boxes provide a form of aggregation. Bagof-boxes can be created by encountering a `bagof/3` program atom:

$$bagof\left(X, B, Y\right) \Rightarrow \textbf{bagof}\left(X;\textbf{and}\left(B;\text{true}\right)_{\{X\}};Y\right)$$

Bagof-boxes are controlled by the *bagof rules*

$$\textbf{and}\left(R, \textbf{bagof}\left(X;\textbf{fail};Y\right), T;\theta\right)_V \Rightarrow \textbf{and}\left(R, T;\theta\wedge Y=[]\right)_V$$

$$\begin{aligned}&\textbf{and}\left(R_1, \textbf{bagof}\left(X;\textbf{or}\left(\ S_1, \textbf{and}\left(;\sigma\right)_W, S_2\ \right);Y\right), R_2;\theta\right)_V \Rightarrow\\ \textbf{and}\left(R_1, \textbf{bagof}\left(C;\textbf{or}\left(\ S_1, S_2\ \right);Y'\right), R_2;\theta\wedge Y=[X'|Y']\wedge\sigma'\right)&_{W\cup V\cup\{X',Y'\}}\end{aligned}$$

where σ is quiet, and σ' is σ with the variables renamed so that X is replaced by X'. The bagof rules do not preserve the order of solutions.

Given a goal G, a AKL computation starts with an and-box $\textbf{and}\left(G;\text{true}\right)_{vars(G)}$ and ends with with either an and-box $\textbf{and}\left(;\theta\right)_{vars(G)\cup vars(\theta)}$ or an or-box
$\textbf{or}\left(\ \textbf{and}\left(;\theta_1\right)_{vars(G)\cup vars(\theta_1)}, \textbf{and}\left(;\theta_2\right)_{vars(G)\cup vars(\theta_2)}, \cdots\ \right)$, each θ_i is a solution of G.

3.4.3 Control

The computation rules described above implicitly contain a hierarchy of control. At the most basic level, rules involving guessing, in this case the nondeterminate promotion rule, should be applied only after all other guess-free rules have been exhausted. The combination of the determinate promotion rule and delaying the nondeterminate promotion rule produces behaviour similar to Andorra Prolog and the BAM, with determinate promotion replacing and-reduction and nondeterminate promotion replacing or-extension.

The use of the conditional operator provides an effect similar to the cut operator of Prolog. Completion of a conditional guard in a clause causes all clauses below that clause to be removed. In addition, any speculative computation that has occurred in the guard is pruned, and the first answer accepted. As an example, consider the program in figure 3.5 which allows two possible choices during nondeterminate

```
p(X, Z) :- q(X, Y) -> Y = Z.

q(X, Y) :- X = a , Y = b  ?  true.
q(X, Y) :- X = a , Y = c  ?  true.
q(X, Y) :- X = b , Y = d  ?  true.
?- p(a, Z)
```

$$\textbf{and}\,(\texttt{p(X, Z)}; X = a)_{\{X,Z\}}$$
$$\Downarrow$$
$$\textbf{and}\left(\textbf{choice}\left(\ \textbf{and}\,(\texttt{q(X, Y)};)_{\{Y\}}\ \texttt{->}\ \texttt{Y = Z}\ \right); X = a\right)_{\{X,Z\}}$$
$$\Downarrow$$
$$\textbf{and}\left(\textbf{choice}\left(\ \textbf{and}\left(\textbf{choice}\left(\begin{array}{l}\textbf{and}\,(\texttt{X = a, Y = b};)\ \texttt{?}\ \ \texttt{true}\\ \textbf{and}\,(\texttt{X = a, Y = c};)\ \texttt{?}\ \ \texttt{true}\\ \textbf{and}\,(\texttt{X = b, Y = d};)\ \texttt{?}\ \ \texttt{true}\end{array}\right);\right)_{\{Y\}}\ \texttt{->}\ \texttt{Y = Z}\ \right); X = a\right)_{\{X,Z\}}$$
$$\Downarrow$$
$$\textbf{and}\left(\textbf{choice}\left(\ \textbf{and}\left(\textbf{choice}\left(\begin{array}{l}\textbf{and}\,(; Y = b)\ \texttt{?}\ \ \texttt{true}\\ \textbf{and}\,(; Y = c)\ \texttt{?}\ \ \texttt{true}\end{array}\right);\right)_{\{Y\}}\ \texttt{->}\ \texttt{Y = Z}\ \right); X = a\right)_{\{X,Z\}}$$
$$\Downarrow$$
$$\textbf{and}\left(\textbf{choice}\left(\ \textbf{or}\left(\begin{array}{l}\textbf{and}\,(; Y = b)_{\{Y\}}\ \texttt{->}\ \texttt{Y = Z}\\ \textbf{and}\,(; Y = c)_{\{Y\}}\ \texttt{->}\ \texttt{Y = Z}\end{array}\right)\ \right); X = a\right)_{\{X,Z\}}$$
$$\Downarrow$$
$$\textbf{and}\left(\textbf{choice}\left(\begin{array}{l}\textbf{and}\,(; Y = b)_{\{Y\}}\ \texttt{->}\ \texttt{Y = Z}\\ \textbf{and}\,(; Y = c)_{\{Y\}}\ \texttt{->}\ \texttt{Y = Z}\end{array}\right); X = a\right)_{\{X,Z\}}$$
$$\Downarrow$$
$$\textbf{and}\left(\textbf{choice}\left(\ \textbf{and}\,(; Y = b)_{\{Y\}}\ \texttt{->}\ \texttt{Y = Z}\ \right); X = a\right)_{\{X,Z\}}$$
$$\Downarrow$$
$$\textbf{and}\,(\texttt{Y = Z}; X = a \wedge Y = b)_{\{X,Y,Z\}}$$
$$\Downarrow$$
$$\textbf{and}\,(; X = a \wedge Y = b \wedge Y = Z)_{\{X,Y,Z\}}$$

Figure 3.5: AKL Computation with Nondeterminate Promotion and Conditional

promotion. When these two solutions are promoted to the parent choice-box, the conditional guard operator eliminates the second choice, producing identical behaviour to a Prolog program where the `->` has been replaced by a `!` .

A similar effect occurs if commit operators are used in place of conditional operators, but with symmetric pruning.

The quietness condition, applied to conditional and commit guarded predicates, ensures that over-eager commitment does not occur. An example of the effect of the quietness condition is:

```
p(a) :- true ?  true.
p(b) :- true ?  true.

q(c) :- true | true.
q(a) :- true | true.

?- p(X), q(X)
```

The configuration produced by the call `q(X)` is
choice (**and** $(; X = c)$ `|  true`, **and** $(; X = a)$ `|  true`). If the quietness condition were not enforced,

one of these clauses would be chosen by the commit rule. The first clause could be promoted, leading to the constraint $X = c$ being promoted and the failure of the computation. Enforcing the quietness condition ensures that the configuration produced from `p(X)` is nondeterministically promoted, and the solution $X = a$ is computed.

3.4.4 Using the AKL

This section describes some of the uses the AKL can be put to. The examples in this section are largely derived from [JH91].

Dependent And-Parallelism

The determinate promotion rule can be used to provide dependent and-parallelism. Figure 3.6 shows the execution path of a simple dependent and-parallel program. In the figure, parts of execution that can be performed in parallel, are. The producer, `p(2, X)` and the consumer, `c(X)` quickly settle down to an alternating pattern, where the consumer is consuming a binding as the producer gets the next binding ready.

Or-Parallelism

Or-parallelism in the AKL is handled by use of nondeterminate promotion and stability. Figure 3.7 shows an example or-parallel computation. After the initial nondeterminate promotion, the remaining two clauses are free to be nondeterminately promoted again.

Nondeterministic Independent And-Parallelism

Nondeterministic independent and-parallelism can be produced by delaying output bindings until nondeterminate promotion has occurred. Several nondeterminate computations are encapsulated by placing them in guards.

```
p(a)  :-  true  ?  true.
p(b)  :-  true  ?  true.
p(c)  :-  true  ?  true.

q(b)  :-  true  ?  true.
q(c)  :-  true  ?  true.
q(d)  :-  true  ?  true.

p1(X)  :-  p(Y)  ?  X = Y.

q1(X)  :-  q(Y)  ?  X = Y.

r(X) :- p1(Y), q1(Z)  ?  X = Y, X = Z.
?-r(X)
```

In the above program the guards of `p1/1` and `q1/1` have local environments that are not restricted by the external environment. The calls to `p/1` and `q/1` are able to independently execute and nondeterministically promote. A join occurs in `r/1` after both `p1(Y)` and `q1(Z)` have completely nondeterministically promoted. The join is implicit in the nondeterminate promotion rule, which occurs when part of a configuration is copied; and-boxes with inconsistent constraints are eliminated by application of the environment synchronisation rule.

```
p(0, X)  :-  true ?  X = [].
p(N, X)  :-  N > 0  ?  X = [N | X1], plus(N1, 1, N) p(N1, X1).

c([])  :-  true ?  true.
c([_ | X])  :-  true ?  true.

?- p(2, X), c(X)
```

`plus(A, B, C)` is defined so that $A + B = C$.

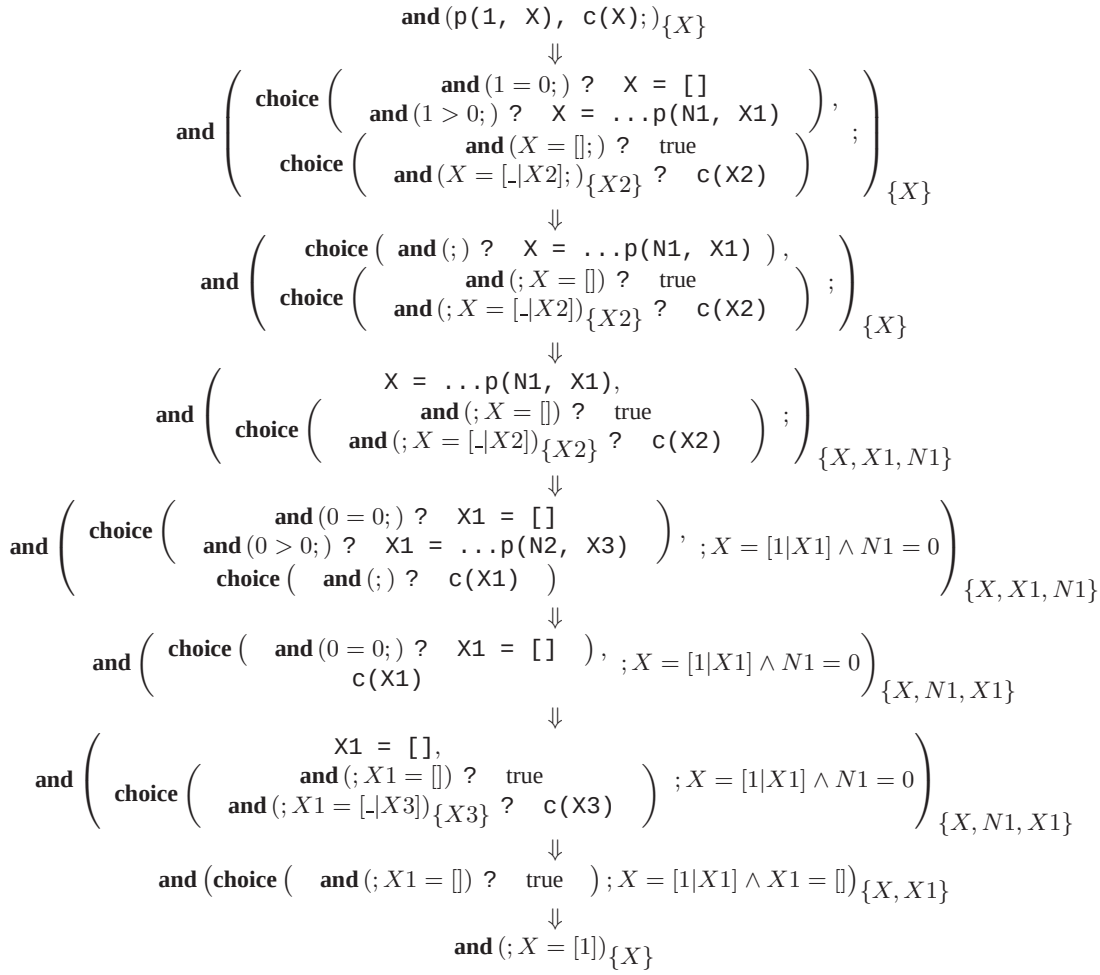

Figure 3.6: Dependent And-Parallelism in the AKL

```
p(X, Y) :- X = a  ?  Y = c.
p(X, Y) :- X = b  ?  Y = b.
p(X, Y) :- X = c  ?  Y = a.

?-p(X)
```

$$\textbf{and}\,(\texttt{p(X, Y)};\text{true})_{\{X,Y\}}$$
$$\Downarrow$$
$$\textbf{and}\left(\textbf{choice}\left(\begin{array}{l}\textbf{and}\,(X=a;\text{true})\ \texttt{?}\ \ \texttt{Y = c}\\ \textbf{and}\,(X=b;\text{true})\ \texttt{?}\ \ \texttt{Y = b}\\ \textbf{and}\,(X=c;\text{true})\ \texttt{?}\ \ \texttt{Y = a}\end{array}\right);\text{true}\right)_{\{X,Y\}}$$
$$\Downarrow$$
$$\textbf{and}\left(\textbf{choice}\left(\begin{array}{l}\textbf{and}\,(;X=a)\ \texttt{?}\ \ \texttt{Y = c}\\ \textbf{and}\,(;X=b)\ \texttt{?}\ \ \texttt{Y = b}\\ \textbf{and}\,(;X=c)\ \texttt{?}\ \ \texttt{Y = a}\end{array}\right);\text{true}\right)_{\{X,Y\}}$$
$$\Downarrow$$
$$\textbf{or}\left(\begin{array}{l}\textbf{and}\,(Y1=c;X1=a)_{\{X1,Y1\}}\\ \textbf{and}\left(\textbf{choice}\left(\begin{array}{l}\textbf{and}\,(;X=b)\ \texttt{?}\ \ \texttt{Y = b}\\ \textbf{and}\,(;X=c)\ \texttt{?}\ \ \texttt{Y = a}\end{array}\right);\text{true}\right)_{\{X,Y\}}\end{array}\right)$$
$$\Downarrow$$
$$\textbf{or}\left(\begin{array}{l}\textbf{and}\,(;X1=a\wedge Y1=c)_{\{X1,Y1\}}\\ \textbf{and}\,(Y2=b;X2=b)_{\{X2,Y2\}}\\ \textbf{and}\left(\textbf{choice}\left(\ \textbf{and}\,(;X=c)\ \texttt{?}\ \ \texttt{Y = a}\ \right);\text{true}\right)_{\{X,Y\}}\end{array}\right)$$
$$\Downarrow$$
$$\textbf{or}\left(\begin{array}{l}\textbf{and}\,(;X1=a\wedge Y1=c)_{\{X1,Y1\}}\\ \textbf{and}\,(;X2=b\wedge Y2=b)_{\{X2,Y2\}}\\ \textbf{and}\,(Y=a;X=c)_{\{X,Y\}}\end{array}\right)$$
$$\Downarrow$$
$$\textbf{or}\left(\begin{array}{l}\textbf{and}\,(;X1=a\wedge Y1=c)_{\{X1,Y1\}}\\ \textbf{and}\,(;X2=b\wedge Y2=b)_{\{X2,Y2\}}\\ \textbf{and}\,(;X=c\wedge Y=a)_{\{X,Y\}}\end{array}\right)$$

Figure 3.7: Or-Parallelism in the AKL

Chapter 4

The AKL and Logic

Since the AKL is intended to be a *logic* programming language, an AKL program is intended to have a logical semantics. This chapter covers the logical structure of the AKL, and introduces a fixpoint semantics for the language that can be used as the concrete semantics for abstract interpretation (see chapter 6). This chapter builds upon the work of Fitting [Fit91] on applying bilattices to logic programming and applies the bilattice model to committed-choice languages in general and the AKL in particular.

4.1 Logical Aspects of the AKL

Before considering the logical aspects of the AKL, it is necessary to provide an intended interpretation of an AKL program. The intended interpretation maps a program P onto a set of logical formulae Σ_P. The AKL execution model should be sound in the sense that any answer computed should be a logical consequence of Σ_P, and complete in the sense that all possible logical consequences of Σ_P are computed.

The basic logical structure of AKL program is given in [Fra94]. A constraint theory, $\mathcal{TC}$, is assumed, with a constraint, σ, holding if $\mathcal{TC} \models \sigma$. Each predicate in the program is given a completed definition:

Definition 4.1.1 The *completed definition*, Σ_P of an AKL program P is given by replacing each predicate `p/n`:

$$\begin{array}{lll} p(\mathbf{x}) & \texttt{:-} & G_1(\mathbf{x},\mathbf{y}) \;\%\; B_1(\mathbf{x},\mathbf{y}) \\ & & \cdots \\ p(\mathbf{x}) & \texttt{:-} & G_m(\mathbf{x},\mathbf{y}) \;\%\; B_m(\mathbf{x},\mathbf{y}) \end{array}$$

by:

$$\forall \mathbf{x} p(\mathbf{x}) \leftrightarrow \left(\begin{array}{c} \exists \mathbf{y_1}(G_1(\mathbf{x},\mathbf{y_1}) \wedge B_1(\mathbf{x},\mathbf{y_1})) \vee \\ \cdots \vee \\ \exists \mathbf{y_m}(G_m(\mathbf{x},\mathbf{y_m}) \wedge B_m(\mathbf{x},\mathbf{y_m})) \end{array} \right)$$

if $\%$ is `?` or `|`. If $\%$ is `->`, the predicate is replaced by:

$$\forall \mathbf{x} p(\mathbf{x}) \leftrightarrow \left(\begin{array}{c} \exists \mathbf{y_1}(G_1(\mathbf{x},\mathbf{y_1}) \wedge B_1(\mathbf{x},\mathbf{y_1})) \vee \\ (\neg \exists \mathbf{y_1}(G_1(\mathbf{x},\mathbf{y_1})) \wedge \exists \mathbf{y_2}(G_2(\mathbf{x},\mathbf{y_2}) \wedge B_2(\mathbf{x},\mathbf{y_2})) \vee \\ \cdots \vee \\ \left(\begin{array}{c} \neg \exists \mathbf{y_1}(G_1(\mathbf{x},\mathbf{y_1})) \wedge \\ \cdots \wedge \\ (\neg \exists \mathbf{y_{m-1}}(G_1(\mathbf{x},\mathbf{y_{m-1}})) \wedge \\ \exists \mathbf{y_m}(G_m(\mathbf{x},\mathbf{y_m}) \wedge B_m(\mathbf{x},\mathbf{y_m}) \end{array} \right) \end{array} \right)$$

The boldface **x, y** and **z** are used to represent vectors of variables.

4.1.1 Negation

Negation as failure is introduced into the AKL by using conditional guards. If `p(X)` is a predicate, then `not p(X)` is given by

```
not_p(X) :- p(X) -> false.
not_p(X) :- true -> true.
```

or more usefully, since we are only interested in seeing whether one branch succeeds,

```
not_p(X) :- some_p(X) -> false.
not_p(X) :- true -> true.

some_p(X) :- p(X) | true.
```

4.1.2 Commit Guards

The presence of the commit operator (`|`) in the AKL tends to produce difficulties with negation. Consider the standard definition for `merge/3`:

```
merge([], B, O) :- | O = B.
merge(A, [], O) :- | O = A.
merge([X | A], B, O) :- | O = [X | O1], merge(A, B, O1).
merge(A, [X | B], O) :- | O = [X | O1], merge(A, B, O1).
```

A query such as `merge([1], [2], [1, 2])` may succeed or fail, depending on the exact path of the computation. Similarly, `merge([1], [2], X), not merge([1], [2], X)` may succeed, breaking the identity $A \wedge \neg A \equiv$ false.

Franzén [Fra94] avoids the difficulties with commit by examining only those predicates with authoritative guards:

Definition 4.1.2 An AKL program, P, is *authoritative* if for every clause of the form

$$H(\mathbf{x}) \quad \texttt{:-} \quad G(\mathbf{x},\mathbf{y}) \mid \ B(\mathbf{x},\mathbf{y},\mathbf{z}).$$

in P

$$\mathcal{TC} \cup \Sigma_P \models G(\mathbf{x},\mathbf{y}) \rightarrow (H(\mathbf{x}) \leftrightarrow \exists \mathbf{z} B(\mathbf{x},\mathbf{y},\mathbf{z}))$$

Authoritative guards guarantee that it does not matter which clause is chosen, in the case of two guards succeeding, as both bodies produce the same results. An example of a predicate with authoritative guards is:

```
min(X, Y, Z) :- X =< Y | Z = X.
min(X, Y, Z) := X >= Y | Z = Y
```

Authoritative guards restrict the commit operator to simply pruning equivalent solutions to the predicate. Unfortunately, `merge/3` is not authoritative, excluding a large class of reactive programs from this logical interpretation of the AKL.

4.1.3 Conditional Guards

Nondeterminate promotion within a conditional guard can produces cases where the guard prunes alternatives within the guard computation that are implied by the completed definition (definition 4.1.1). As an example, consider the program:

```
p :- r(X) -> X = a.

r(X) :- ?  X = b.
r(X) :- ?  X = a.
```

In this case, `p/0` will fail, as the second solution to `r(X)` is pruned. However, the completed definition of `r/1` is $\forall X r(X) \leftrightarrow (X = b \vee X = a)$ and therefore that of `p/0` is $p \leftrightarrow$ true. The solution adopted by Franzén is to restrict the set of programs examined to those with indifferent guards:

Definition 4.1.3 An AKL program, P, is *indifferent* if, for every clause of the form

$$H(\mathbf{x}) \quad \texttt{:-} \quad G(\mathbf{x},\mathbf{y}) \texttt{ -> } \; B(\mathbf{x},\mathbf{y},\mathbf{z}).$$

in P,

$$\mathcal{TC} \cup \Sigma_P \models G(\mathbf{x},\mathbf{y}) \wedge G(\mathbf{x},\mathbf{w}) \rightarrow (\exists \mathbf{z} B(\mathbf{x},\mathbf{y},\mathbf{z}) \leftrightarrow B(\mathbf{x},\mathbf{w},\mathbf{z}))$$

Most AKL programs are intended to be indifferent, although an insistence on indifference eliminates `once/1`-style predicates written with conditional guards, which are intended to accept the first solution to a query. Such predicates can be used to accept a single permutation of some set of values, such as a register allocation. However these predicates can be re-cast as commit-guarded predicates without damaging the intended behaviour of the predicate.

4.1.4 Recursive Guards

Recursive guards present a problem in the AKL, as it is possible for a program to enter an infinite loop, even though the logical semantics for the program have a definite value. As an example, consider the predicate `p/1`:

```
p(0).
p(s(X)) :- p(X) ? true.
```

Calling this program with `?-p(s(s(s(0))))` will result in a response of `true`. However calling this program with the query `?-p(X)` will result in an infinite loop, as the recursive guard is tried within a new environment with each local fork. Since the AKL always expands guards (a deterministic operation) in preference to nondeterminate promotion, this query will never terminate.

Generally, recursive guards should be avoided in the AKL; they produce infinite loops and make the program unstratified, since a recursive call within a conditional guard results in a recursive form of negation. For this reason only guard-stratified programs are used.

Definition 4.1.4 A program P is *guard-stratified* if it can be partitioned into programs $P_1, \ldots, P_l$ with $P_1 \cup \cdots \cup P_l = P$ such that each guard of each clause in P_k contains only primitive constraints, or atoms defined by clauses in $P_1 \cup \cdots \cup P_{k-1}$

4.2 A Bilattice Interpretation of AKL Programs

The introduction of authoritative AKL programs weakens the number of uses that a logical interpretation of AKL can be put to. Authoritative guards are intended to eliminate the situations where a query can either succeed or fail, keeping the logical definition within the bounds of a classical two-valued logic, with success mapping onto true and failure onto false.

An alternative to insisting on authoritative guards is to move beyond two-valued logic, and introduce a multi-valued logic which contains a value that maps on to the notion of “sometimes succeeds — sometimes fails.”

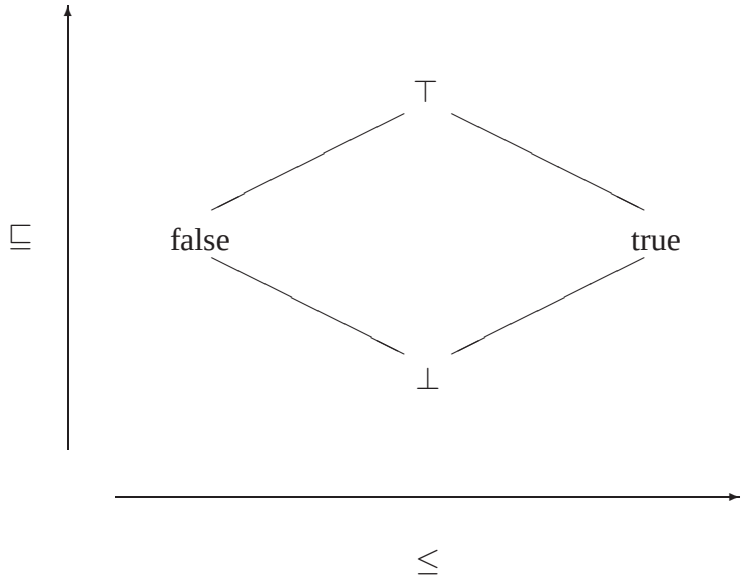

Figure 4.1: The logic **FOUR**

4.2.1 Bilattices

Bilattices were introduced to logic programming by Fitting [Fit91] with the intention of modelling uncertain and unknown information. The characterisation of bilattices given here is taken from [Fit91], with some changes in notation[1].

Definition 4.2.1 An *interlaced bilattice* is a set, B, together with two partial orderings, $\leq$ and $\sqsubseteq$, such that:

1. each of $\leq$ and $\sqsubseteq$ gives B the structure of a complete lattice.

2. the lub and glb operations for each partial ordering are monotone with respect to the other partial ordering.

4.2.2 A Logic Based on FOUR

The AKL, needs the simplest form of bilattice, **FOUR**[Bel77]. The Hasse diagram for **FOUR** is shown in figure 4.1.

For the purposes of the AKL, $\top$ is interpreted as meaning "both succeeds and fails" and $\bot$ has a meaning of undefined, which can be interpreted as either "still being computed," "infinite loop" or "deadlocked," depending on context. The values true and false are interpreted as "always succeeds" and "always fails." With these meanings, $\sqsubseteq$ can be interpreted as an ordering on the range of possible results a program can give.

The standard Boolean operations of logical and ($\wedge$), or ($\vee$) and not ($\neg$) now have a set of dual operations based on the $\sqsubseteq$ ordering: indecisive-and ($\sqcap$), indecisive-or ($\sqcup$) and indecisive-not ($-$) [2]. Using the **FOUR** bilattice, the standard Boolean operators, and their duals, are given in table 4.1. The four basic operators are derived from $a \wedge b \equiv \text{glb}_{\leq}(a, b)$, $a \vee b \equiv \text{lub}_{\leq}(a, b)$, $a \sqcap b \equiv \text{glb}_{\sqsubseteq}(a, b)$ and $a \sqcup b \equiv \text{lub}_{\sqsubseteq}(a, b)$. Negation is a reflection about the $\sqsubseteq$ axis and indecisive negation a reflection about the $\leq$ axis. Implication is defined by $a \rightarrow b \equiv \neg - a \vee b$ and equivalence by $a \leftrightarrow b \equiv (a \rightarrow b) \wedge (a \leftarrow b)$.

The quantification operators can be interpreted as conjunction or disjunction over the domain of a variable: $\exists x a(x) \equiv \bigvee_{y \in dom(x)} a(y)$ and $\forall x a(x) \equiv \bigwedge_{y \in dom x} a(y)$ [3].

[1] In this thesis, $\leq$ is used instead of the $\leq_t$ in [Fit91]. Similarly, $\sqsubseteq$ is used in place of $\leq_k$, $\sqcap$ in place of $\otimes$ and $\sqcup$ in place of $\oplus$.

[2] These names derive from the idea that indecisive-or can't make up its mind about true$\sqcup$false; the initial motivation for introducing

a	b	$a \wedge b$	$a \vee b$	$a \sqcap b$	$a \sqcup b$	$\neg a$	$-a$	$a \rightarrow b$	$a \leftrightarrow b$
true	true	true	true	true	true	false	true	true	true
true	$\top$	$\top$	true	true	$\top$	false	true	$\top$	$\top$
true	$\bot$	$\bot$	true	$\bot$	true	false	true	$\bot$	$\bot$
true	false	false	true	$\bot$	$\top$	false	true	false	false
$\top$	true	$\top$	true	true	$\top$	$\top$	$\bot$	true	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\bot$	true	true
$\top$	$\bot$	false	true	$\bot$	$\top$	$\top$	$\bot$	$\bot$	false
$\top$	false	false	$\top$	false	$\top$	$\top$	$\bot$	$\bot$	$\bot$
$\bot$	true	$\bot$	true	$\bot$	true	$\bot$	$\top$	$\top$	$\bot$
$\bot$	$\top$	false	true	$\bot$	$\top$	$\bot$	$\top$	true	false
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\top$	$\top$	true
$\bot$	false	false	$\bot$	$\bot$	false	$\bot$	$\top$	true	$\top$
false	true	false	true	$\bot$	$\top$	true	false	true	false
false	$\top$	false	$\top$	false	$\top$	true	false	true	$\bot$
false	$\bot$	false	$\bot$	$\bot$	false	true	false	true	$\top$
false	false	false	false	false	false	true	false	true	true

Table 4.1: Boolean operators for **FOUR**

The combination of **FOUR** and the operations defined above form an extended Boolean algebra, satisfying the normal identities of a Boolean algebra. These identities are summarised in table 4.2; most are consequences of the properties of interlaced bilattices (see [Fit91]).

4.2.3 Commit Predicates

Armed with the bilattice described above, it is now possible to describe a logical interpretation of a commit-guarded predicate. If two guards can potentially succeed and one body succeeds and the other body fails, then the truth-values of these two clauses can be combined via indecisive-or into a truth-value of $\top$.

Intuitively the failure of a guard should give the clause a truth-value of $\bot$. This means that the clause will not appear in the truth-value of the predicate. A guard operator should have the following properties:

$$a \| b = \begin{cases} \bot & \text{if } a = \bot \text{ or } a = \text{false} \\ b & \text{otherwise} \end{cases}$$

This guard operator can be defined as $a \| b \equiv (\neg a \sqcup \text{true}) \sqcap (a \sqcup \text{false}) \sqcap b$.

All predicates in the AKL follow the negation as failure rule; if no clause succeeds, then a truth value of false is assumed. As a special case, all guard operators failing is also assumed to mean a truth value of false. With the above definition of $a \| b$, all guards failing will yield a truth value of $\bot$. While this truth value may be aesthetically satisfying it does not match the definition of the language, and an explicit term for failure needs to be introduced. If a predicate is defined by a set of clauses,

$$H(\mathbf{x}) \quad \texttt{:-} \quad G_i(\mathbf{x}, \mathbf{y}) \mid \ B_i(\mathbf{x}, \mathbf{y}, \mathbf{z}), 1 \leq i \leq n,$$

then the additional term for failure can be defined as:

$$fail(G_1, \ldots, G_n) = \begin{cases} \text{false} & \text{if all } G_i \text{ are false or } \top \\ \bot & \text{otherwise} \end{cases}$$

$fail(G_1, \ldots, G_n) \equiv G_1 \sqcap \cdots \sqcap G_m \sqcap \text{false}$ matches this definition.

a multi-valued logic. In [Fit91] indecisive-not is called confluence.

[3] Similar quantification operators can be defined for the indecisive operators. These operators play no part in the fixpoint semantics for the AKL and are therefore ignored.

Null	
$a \wedge \text{false} = \text{false}$	$a \vee \text{true} = \text{true}$
$a \sqcap \bot = \bot$	$a \sqcup \top = \top$
Identity	
$a \wedge \text{true} = a$	$a \vee \text{false} = a$
$a \sqcap \top = a$	$a \sqcup \bot = a$
Idempotent	
$a \wedge a = a$	$a \vee a = a$
$a \sqcap a = a$	$a \sqcup a = a$
Inverse	
$\neg\neg a = a$	$--a = a$
$a \wedge \neg -a = \text{false}$	$a \vee \neg -a = \text{true}$
$a \sqcap \neg -a = \bot$	$a \sqcup \neg -a = \top$
Commutative	
$a \wedge b = b \wedge a$	$a \vee b = b \vee a$
$a \sqcap b = b \sqcap a$	$a \sqcup b = b \sqcup a$
Associative	
$(a \wedge b) \wedge c = a \wedge (b \wedge c)$	$(a \vee b) \vee c = a \vee (b \vee c)$
$(a \sqcap b) \sqcap c = a \sqcap (b \sqcap c)$	$(a \sqcup b) \sqcup c = a \sqcup (b \sqcup c)$
Distributive	
$a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$	$a \wedge (b \sqcap c) = (a \wedge b) \sqcap (a \wedge c)$
$a \wedge (b \sqcup c) = (a \wedge b) \sqcup (a \wedge c)$	
$a \vee (b \wedge c) = (a \vee b) \wedge (a \vee c)$	$a \vee (b \sqcap c) = (a \vee b) \sqcap (a \vee c)$
$a \vee (b \sqcup c) = (a \vee b) \sqcup (a \vee c)$	
$a \sqcap (b \wedge c) = (a \sqcap b) \wedge (a \sqcap c)$	$a \sqcap (b \vee c) = (a \sqcap b) \vee (a \sqcap c)$
$a \sqcap (b \sqcup c) = (a \sqcap b) \sqcup (a \sqcap c)$	
$a \sqcup (b \wedge c) = (a \sqcup b) \wedge (a \sqcup c)$	$a \sqcup (b \vee c) = (a \sqcup b) \vee (a \sqcup c)$
$a \sqcup (b \sqcap c) = (a \sqcup b) \sqcap (a \sqcup c)$	
Absorption	
$a \wedge (a \vee b) = a$	$a \vee (a \wedge b) = a$
$a \sqcap (a \sqcup b) = a$	$a \sqcup (a \sqcap b) = a$
DeMorgan's	
$\neg(a \wedge b) = \neg a \vee \neg b$	$\neg(a \vee b) = \neg a \wedge \neg b$
$\neg(a \sqcap b) = \neg a \sqcap \neg b$	$\neg(a \sqcup b) = \neg a \sqcup \neg b$
$-(a \wedge b) = -a \wedge -b$	$-(a \vee b) = -a \vee -b$
$-(a \sqcap b) = -a \sqcup -b$	$-(a \sqcup b) = -a \sqcap -b$

Table 4.2: Boolean identities on **FOUR**

FOUR can now be used to give a logical interpretation of the AKL. The interpretations of `?` and `->` guarded predicates are the same as section 4.1. A set of commit-guarded clauses,

$$H(\mathbf{x}) \quad \texttt{:-} \quad G_i(\mathbf{x},\mathbf{y}) \mid \ B_i(\mathbf{x},\mathbf{y},\mathbf{z}), 1 \leq i \leq n,$$

now has the logical interpretation:

$$\forall \mathbf{x} H(\mathbf{x}) \leftrightarrow \left(\begin{array}{c} \exists \mathbf{y_1}(G_1(\mathbf{x},\mathbf{y_1}) \| B_1(\mathbf{x},\mathbf{y_1})) \sqcup \\ \cdots \sqcup \\ \exists \mathbf{y_m}(G_m(\mathbf{x},\mathbf{y_m}) \| B_m(\mathbf{x},\mathbf{y_m})) \sqcup \\ (\exists \mathbf{y_1} G_1(\mathbf{x},\mathbf{y_1}) \sqcap \cdots \sqcap \exists \mathbf{y_m} G_m(\mathbf{x},\mathbf{y}) \sqcap \text{false}) \end{array} \right)$$

4.3 A Fixpoint Semantics for the AKL

A fixpoint semantics for the AKL can now be derived in terms of some suitable immediate consequences operator. The treatment here follows that of [Fit91].

Definition 4.3.1 An *interpretation* is a mapping v from a subset of ground atomic formulae to **FOUR**. The interpretation is *in* the constraint theory $\mathcal{TC}$ provided that for each constraint θ, if $\mathcal{TC} \models \theta$ then $v\theta = \text{true}$ and if $\mathcal{TC} \not\models \theta$ then $v\theta = \text{false}$. $\mathcal{I}$ denotes the domain of interpretations.

It may be possible to extend the constraint theory to handle the full four values of **FOUR**. However, the proofs of the soundness and completeness theorems (theorems 4.4.1 and 4.4.2) require constraints to only have two values.

Throughout this section functional notation is used to reduce the number of parentheses; $v\,\theta$ is interpreted as the application of the mapping v to the argument θ.

Definition 4.3.2 The *composition* of two interpretations, v_1 and v_2, denoted by $v_1 \succ v_2$, is defined as

$$(v_1 \succ v_2)\, A = \begin{cases} v_1\, A & \text{if } A \in dom(v_1) \\ v_2\, A & \text{otherwise} \end{cases}$$

Definition 4.3.3 Given two interpretations v_1 and v_2, $v_1 \leq v_2$ if for all closed atomic formulae $A \in dom(v_1)$, $v_1 A \leq v_2 A$. A similar definition holds for $\sqsubseteq$.

Definition 4.3.4 If v is an interpretation and ϕ a closed formula, then $v\phi$ is constructed by interpreting $\wedge$ in ϕ as $\wedge$ on **FOUR**, $\vee$ in ϕ as $\vee$ on **FOUR**, etc.

An immediate consequences operator for an AKL program can now be defined. The immediate consequences operator is intended for use on guard stratified program. The operator takes two arguments: the first argument is the interpretation provided by previous stratifications, the second is the current interpretation.

Definition 4.3.5 Let P be an AKL program. $\Phi_P : \mathcal{I} \to \mathcal{I} \to \mathcal{I}$ is defined as follows:

1. If $H(\mathbf{x}) \quad \texttt{:-} \quad G_i(\mathbf{x},\mathbf{y})\,\texttt{?} \quad B_i(\mathbf{x},\mathbf{y},\mathbf{z}) \in P$ then

$$\Phi_P\, p\, v\, H(\mathbf{x}) = (p \succ v) \left(\begin{array}{c} \exists \mathbf{y_1}, \mathbf{z_1} G_1(\mathbf{x},\mathbf{y_1}) \wedge B_1(\mathbf{x},\mathbf{y_1},\mathbf{z_1}) \vee \\ \cdots \vee \\ \exists \mathbf{y_m}, \mathbf{z_m} G_m(\mathbf{x},\mathbf{y_m}) \wedge B_m(\mathbf{x},\mathbf{y_m},\mathbf{z_m}) \end{array} \right)$$

2. If $H(\mathbf{x})$: - $G_i(\mathbf{x},\mathbf{y})$ -> $B_i(\mathbf{x},\mathbf{y},\mathbf{z}) \in P$ then

$$\Phi_P\, p\, v\, H(\mathbf{x}) = (p \succ v)\left(\begin{array}{c} \exists \mathbf{y_1}, \mathbf{z_1} G_1(\mathbf{x},\mathbf{y_1}) \wedge B_1(\mathbf{x},\mathbf{y_1},\mathbf{z_1}) \vee \\ \cdots \vee \\ \left(\begin{array}{l} \neg\exists \mathbf{y_1} G_1(\mathbf{x},\mathbf{y_1}) \wedge \\ \ldots \wedge \\ \neg\exists \mathbf{y_{m-1}} G_{m-1}(\mathbf{x},\mathbf{y_{m-1}}) \wedge \\ \exists \mathbf{y_m}, \mathbf{z_m} G_m(\mathbf{x},\mathbf{y_m}) \wedge B_m(\mathbf{x},\mathbf{y_m},\mathbf{z_m}) \end{array}\right) \end{array}\right)$$

3. If $H_i((\mathbf{x}))$: - $G_i(\mathbf{x},\mathbf{y})$ | $B_i(\mathbf{x},\mathbf{y},\mathbf{z}) \in P$ then

$$\Phi_P\, p\, v\, H(\mathbf{x}) = (p \succ v)\left(\begin{array}{c} \exists \mathbf{y_1}, \mathbf{z_1} G_1(\mathbf{x},\mathbf{y_1}))\|B_1(\mathbf{x},\mathbf{y_1},\mathbf{z_1}) \sqcup \\ \cdots \sqcup \\ \exists \mathbf{y_m}, \mathbf{z_m} G_m(\mathbf{x},\mathbf{y_m}))\|B_m(\mathbf{x},\mathbf{y_m},\mathbf{z_m}) \sqcup \\ (\exists \mathbf{y_1} G_1(\mathbf{x},\mathbf{y_1}) \sqcap \cdots \sqcap \exists \mathbf{y_m} G_m(\mathbf{x},\mathbf{y_m}) \sqcap \text{false}) \end{array}\right)$$

4. $\Phi_P\, p\, v\, \phi = (p \succ v)\phi$ otherwise.

Definition 4.3.6 Given P, an indifferent, guard stratified program , with Q as one of the stratifications, and an interpretation, p, then:

$\Phi_Q\, p \uparrow 0 = p$.
$\Phi_Q\, p \uparrow \alpha = \Phi_Q\, p(\Phi_Q\, p \uparrow (\alpha - 1))$, if α is a successor ordinal.
$\Phi_Q\, p \uparrow \alpha = lub\{\Phi_Q\, p \uparrow \beta : \beta < \alpha\}$, if α is a limit ordinal.

Proposition 4.3.1 Suppose P is an indifferent, guard stratified program, partitioned into sub-programs $\{P_1, \ldots, P_m\}$. Further suppose that there exists an interpretation, p, which is in $\mathcal{TC}$, and has a domain covering all guards in $P_i = Q$. Then $\Phi_Q\, p$ is continuous.

Proof The proof is a suitable variation of that in [Llo84]. $\Phi_Q\, p$ is continuous if $lub(\Phi_Q\, p\, X) = \Phi_Q\, p\, \text{lub}(X)$ for all directed sets X.

Since both $\leq$ and $\sqsubseteq$ are defined in a pointwise fashion on interpretations (definition 4.3.3), it is clear that $\text{lub}(X)\, \phi = \text{lub}(X\, \phi)$ for all formulae $\phi \in \bigcup dom(X)$ for both orderings.

$\Phi_Q\, p\, \text{lub}(X)\, H = t,\ t \in$ **FOUR**
iff $H \leftrightarrow \phi \in \Sigma_Q$ and $\text{lub}(X)\, \phi = t$
iff $H \leftrightarrow \phi \in \Sigma_Q$ and $\text{lub}(X\, \phi) = t$
iff $\text{lub}(\Phi_Q\, p\, X\, H) = t$ □

Corollary 4.3.1 $\Phi_Q\, p$ has a least fixpoint and greatest fixpoint, with $\text{lfp}(\Phi_Q\, p) = \Phi_Q\, p \uparrow \omega$. See [Llo84].

Theorem 4.3.1 (Fixpoint characterisation of the AKL)

Suppose P is an indifferent, guard stratified program, partitioned into sub-programs $\{P_1, \ldots, P_m\}$. Further suppose that p is a model in $\mathcal{TC}$ for $P_1 \cup \cdots \cup P_{i-1}$. Then $\Phi_{P_i}\, p \uparrow \omega$ is a model for $\mathcal{TC} \cup P_1 \cup \cdots \cup P_i$.

Proof This proof is, again, a variation on [Llo84]. To start with, v is a model for P_i if $\Phi_{P_i}\, p\, v \sqsubseteq v$ since v is a model for P_i if, for each $H \leftrightarrow \phi \in \Sigma_{P_i}$, $v\, H = v\, \phi$. Since $\Phi_{P_i}\, p \uparrow \omega = \text{lfp}(\Phi_{P_i}\, p)$ it follows that $\Phi_{P_i}\, p\, (\Phi_{P_i}\, p \uparrow \omega) \sqsubseteq \Phi_{P_i}\, p \uparrow \omega$. □

4.4 The AKL Execution Model

The fixpoint semantics for the AKL derived in section 4.3 provide a suitable model for the intended meaning of an AKL program. However, the AKL itself follows a top-down execution model and it becomes necessary to derive the relationship between the fixpoint semantics and the execution model.

The AKL execution model consists of a series of transitions over configurations. If A and B are configurations (as defined in section 3.4) then $A \Rightarrow^t B$ is a derivation from A to B via the execution rule t. If the execution rule is obvious from the context, then $A \Rightarrow B$ is a suitable shorthand. Multiple unspecified transitions, can be written as $A \Rightarrow^* B$. A configuration A is a *terminal configuration* if there exists no admissible transition $A \Rightarrow B$. A terminal configuration A is an *end configuration* if A has the form **choice** (**and** $(;\theta_1)_{\text{vars}(\theta_1)}$ `?` `true`, ..., **and** $(;\theta_n)_{\text{vars}(\theta_n)}$ `?` `true`). Terminal configurations that are not end configurations are *deadlock configurations*.

A logical mapping for AKL configurations is also necessary. Following the model of sections 4.1 and 4.2.3 each box can be mapped onto a logical interpretation in a manner similar to that of [Fra94], but adjusted to the new definition for commit.

Definition 4.4.1 The *logical interpretation* of a configuration C, written as C^*, is defined recursively as:

1. **fail**$^* =$ false.
2. $A^* = A$, where A is an atom or primitive constraint. In particular, `true`$^* =$ true.
3. $(A, B)^* = A^* \wedge B^*$
4. **and** $(R;\theta)^*_V = \exists V(R^* \wedge \theta)$.
5. **choice** (G_1 `?` $B_1, \ldots, G_n$ `?` B_n)$^* = G_1^* \wedge B_1^* \vee \cdots \vee G_n^* \wedge B_n^*$.
6. **choice** (G_1 `->` $B_1, \ldots, G_n$ `->` B_n)$^* = G_1^* \wedge B_1^* \vee \cdots \vee \neg G_1^* \wedge \cdots \wedge \neg G_{n-1}^* \wedge G_n^* \wedge B_n^*$.
7. **choice** (G_1 `|` $B_1, \ldots, G_n$ `|` B_n)$^* = G_1^* \| B_1^* \sqcup \cdots \sqcup G_n^* \| B_n^* \sqcup (G_1^* \sqcap \cdots \sqcap G_n^* \sqcap \text{false})$.
8. **or** ($C_1, \ldots, C_n$)$^* = C_1^* \vee \cdots \vee C_n^*$

Lemma 4.4.1 Given a program P with model v and an initial goal G, and the transition sequence $A =$ **choice** (**and** $(G;\text{true})_{vars(G)}$ `?` `true`) $\Rightarrow^* B$ then $vB^* \sqsubseteq v\exists G$.

Proof The proof proceeds by induction on the number of transitions. Clearly, for zero transitions, $vA^* = v\exists G$. Suppose that the proposition holds for $A \Rightarrow^* C$ and the next transition in the sequence is $C \Rightarrow^t B$, then each transition can be examined:

In the cases of constraint imposition, environment synchronisation, failure propagation, determinate promotion and non-determinate promotion, the transition is a re-arrangement of the expression C^* following the identity rules of **FOUR**. Obviously $vB^* = vB^*$.

In the case of local forking, the transition is
$H \Rightarrow$ **choice** (**and** $(G_1;\text{true})_{vars(G_1)}$ `%` $B_1, \ldots,$ **and** $(G_n;\text{true})_{vars(G_n)}$ `%` B_n) for some H. Since $H \leftrightarrow \phi \in \Sigma_P$ has ϕ given by definitions in 4.1.1 and section 4.2.3, and these are the same as the definition given in definition 4.4.1, it must be that $vC^* = vB^*$.

In the case of an application of the cut rule, there exists a final guard $G =$ **and** $(;\theta)_V$ and $vG =$ true since $\mathcal{TC} \models \theta$. In this case all subsequent guards have $\neg G$ in them, and must therefore have $v\neg G =$ false. From the identity rules of **FOUR**, $vC^* = vB^*$.

If a commit rule is applied, then all surrounding $\sqcup$ parts of the expression in C^* are removed. Since $\sqcup$ is defined in terms of lub on $\sqsubseteq$ it follows that $C^* \sqsubseteq B^*$. □

Lemma 4.4.1 can be used to provide a soundness theorem for the AKL:

Theorem 4.4.1 (Soundness of AKL) If P is an indifferent, stratified AKL program, G a goal and **choice** (**and** $(G; \text{true})_{vars(G)}$ `?  true`) $\Rightarrow^*$ B where B is an end configuration, then $\Sigma_P \cup \mathcal{TC} \models B^* \sqsubseteq \exists G$

Proof By application of lemma 4.4.1 □

In theorem 4.4.1, B will be a collection of constraints which are modelled by $\mathcal{TC}$. The $\sqsubseteq$ is necessary, rather than implication, since G may have a truth value of $\top$ and true $\rightarrow \top$ has a truth value of $\top$.

Theorem 4.4.1 also serves as a proof of soundness of failure, since the end configuration may be **fail**.

Completeness is a little more difficult for AKL programs. Assuming that the program actually terminates, then it is not possible for completeness to be meaningfully defined for programs that have predicates with a truth value of $\top$. However completeness is possible for the subset of authoritative AKL programs and queries that terminate.

Theorem 4.4.2 (Completeness of AKL) If P is an authoritative, indifferent, guard stratified AKL program, G a goal and **choice** (**and** $(G; \text{true})_{vars(G)}$ `?  true`) $\Rightarrow^*$ B where B is an end configuration then $\Sigma_P \cup \mathcal{TC} \models \exists G \leftrightarrow B^*$

Proof If P is authoritative, then using definition 4.1.2, it is not possible for G to have a truth value of $\top$. B consists only of constraints, and constraints must have a truth value of true or false. Since $\Sigma_P \cup \mathcal{TC} \models B^* \sqsubseteq \exists G$, B^* is true or false and $\exists G$ is either true, false or $\bot$ then $\exists G$ is either true or false and therefore, from the Hasse diagram of **FOUR** $\Sigma_P \cup \mathcal{TC} \models G \leftrightarrow B^*$. □

The above theorem also shows the relationship between the logical semantics derived by the fixpoint construction given here and the semantics given in [Fra94]. Provided a program is authoritative, indifferent and guard stratified, no goal has a truth-value of $\top$ and (with the exception of undefined goals — those having a truth value of $\bot$) the true and false truth values must match.

Since there is a value of $\bot$ for undefined truth values, a more accurate model of the truth value of an AKL configuration can be derived by giving unevaluated atoms a truth value of $\bot$.

Definition 4.4.2 The *execution interpretation* of a configuration C, written as C^+ is the same as that of the logical interpretation of C, replacing all instances of $*$ by $+$ with the exception of case 2. For case 2 $A^+ = \bot$ if A is an atom or primitive constraint.

Lemma 4.4.2 Given a program P with model v and a configuration C, then $v(C^+) \sqsubseteq v(C^*)$.

Proof Both C^* and C^+ are monotone with respect to their sub-components. This is clearly true in all cases except case 7 in definition 4.4.1, since the $\wedge$ and $\vee$ operations are monotone on $\sqsubseteq$. In case 7, the $\sqcup$ operator builds a lub of its arguments and is therefore also monotone on $\sqsubseteq$.

Since the difference between C^* and C^+ is that C^+ replaces any atom A with $\bot$ instead of $v(A)$ and $\bot \sqsubseteq v(A)$ it follows that $C^+ \sqsubseteq C^*$. □

4.5 Some Examples

This section gives some examples of the Φ_P operator acting on programs with various degrees of well-behavedness. For convenience, Φ_{P_i} is used instead of $\Phi_{P_i}(\Phi_{P_{i-1}} \uparrow \omega)$.

```
p(X, Y)  :-  X = a  ?  Y = b.  q(X, Y)  :-  p(X, Y)  -> true.
p(X, Y)  :-  X = b  ?  Y = a.  q(X, Y)  :-  p(X, Z)  -> q(Z, Y).
```

$$P_1 = \{\texttt{p/2}\}$$

$$\Phi_{P_1} \uparrow 0 = \{p(X,Y) \leftrightarrow \bot\}$$
$$\Phi_{P_1} \uparrow 1 = \{p(X,Y) \leftrightarrow (X = a \wedge Y = b) \vee (X = b \wedge Y = a)\}$$
$$\Phi_{P_1} \uparrow \omega = \Phi_{P_1} \uparrow 1$$

$$P_2 = \{\texttt{q/2}\}$$

$$\Phi_{P_2} \uparrow 0 = \Phi_{P_1} \uparrow \omega \cup \{q(X,Y) \leftrightarrow \bot\}$$
$$\Phi_{P_2} \uparrow 1 = \Phi_{P_1} \uparrow \omega \cup \left\{ q(X,Y) \leftrightarrow \left(\begin{array}{c} (X = a \wedge Y = b) \vee (X = b \wedge Y = a) \vee \\ \left(\begin{array}{c} (\neg(X = a \wedge Y = b) \vee (X = b \wedge Y = a)) \wedge \\ \wedge (X = a \vee X = b) \wedge \bot \end{array} \right) \end{array} \right) \right\}$$
$$\Phi_{P_2} \uparrow 2 = \Phi_{P_1} \uparrow \omega \cup \left\{ q(X,Y) \leftrightarrow \left(\begin{array}{c} (X = a \wedge Y = b) \vee \\ (X = b \wedge Y = a) \vee \\ (X = a \wedge Y = a) \vee \\ (X = b \wedge Y = b) \end{array} \right) \right\}$$
$$\Phi_{P_2} \uparrow \omega = \Phi_{P_2} \uparrow 2$$

Figure 4.2: A Guard Stratified, Indifferent, Authoritative Program

```
p(X, Y)  :-  X = a  | Y = b.
p(X, Y)  :-  X = a  | Y = a.

q  :-  p(a, b).
```

$$P_1 = \{\texttt{p/2}, \texttt{q/0}\}$$

$$\Phi_{P_1} \uparrow 0 = \{p(X,Y) \leftrightarrow \bot, q \leftrightarrow \bot\}$$
$$\Phi_{P_1} \uparrow 1 = \left\{ p(X,Y) \leftrightarrow \left(\begin{array}{c} ((X \neq a \sqcup \text{true}) \sqcap (X = a \sqcup \text{false}) \sqcap Y = b \sqcup \\ ((X \neq a \sqcup \text{true}) \sqcap (X = a \sqcup \text{false}) \sqcap Y = a \sqcup \\ (X = a \sqcap \text{false}) \end{array} \right), q \leftrightarrow \bot \right\}$$
$$\Phi_{P_1} \uparrow 2 = \left\{ p(X,Y) \leftrightarrow \left(\begin{array}{c} ((X \neq a \sqcup \text{true}) \sqcap (X = a \sqcup \text{false}) \sqcap Y = b \sqcup \\ ((X \neq a \sqcup \text{true}) \sqcap (X = a \sqcup \text{false}) \sqcap Y = a \sqcup \\ (X = a \sqcap \text{false}) \end{array} \right), q \leftrightarrow \top \right\}$$
$$\Phi_{P_1} \uparrow \omega = \Phi_{P_1} \uparrow 2$$

Figure 4.3: A Guard Stratified, Indifferent Program

```
p  :-  p

q(X)  :-  p  | X = a.
q(X)  :-  true | X = b.
```

$$P_1 = \{\texttt{p/0}\}$$

$$\Phi_{P_1} \uparrow 0 = \{p \leftrightarrow \bot\}$$
$$\Phi_{P_1} \uparrow \omega = \Phi_{P_1} \uparrow 0$$

$$P_2 = \{\texttt{q/1}\}$$

$$\Phi_{P_2} \uparrow 1 = \Phi_{P_1} \uparrow \omega \cup \{q(X) \leftrightarrow \bot\}$$
$$\Phi_{P_2} \uparrow 1 = \Phi_{P_1} \uparrow \omega \cup \{q(X) \leftrightarrow X = b\}$$
$$\Phi_{P_2} \uparrow \omega = \Phi_{P_2} \uparrow 1$$

Figure 4.4: A Guard Stratified, Indifferent Program with Looping

4.5.1 Well-Behaved Programs

Well-behaved programs are guard stratified and indifferent. An example well-behaved program, which is also authoritative, is shown in figure 4.2. In this program `q/2` computes the transitive closure of `p/2`, with a conditional guard to ensure that `q/2` is used for queries only.

An example non-authoritative, well-behaved program is shown in figure 4.3. In this case a race condition exists between the two clauses of `p/2`, which will result in a query to `q/0` either succeeding or failing. As a result `q/0` has a truth value of $\top$.

Another example of a well-behaved program is a program with an infinite loop in it. In figure 4.4, `p/0` has no single truth value and so retains a truth value of $\bot$. If this truth value is fed into the guard of `q/1` then the other clause completely describes the predicate. Operationally, if the computation rule is fair, the second guard of `q/2` will succeed before the first guard and prune the first guard away.

The fixpoint for the semantics does not necessarily perfectly reflect the actual behaviour of an AKL program. Clearly if the commit rule prunes unfairly then some of the success configurations implied by the fixpoint semantics will never be reached. The fixpoint semantics take no account of modes, which can lead to certain unreachable configurations being defined as true. For example, the program

```
p(X, Y)  :-  X = a  | Y = b.
p(X, Y)  :-  X = b  | Y = a.

q(X, Y)  :-  true  | Y = a.
q(X, Y)  :-  X = a  | Y = b.

r(X) :- p(X, Y), q(Y, X).
```

has a fixpoint of

$$\left\{ \begin{array}{l} p(X,Y) \leftrightarrow (X = a \| Y = b) \sqcup (X = a \| Y = b) \sqcup (X = a \sqcap X = b \sqcap \text{false}) \\ q(X,Y) \leftrightarrow Y = a \sqcup (X = a \| Y = b) \\ r(X) \leftrightarrow (X = a \sqcup X = b) \end{array} \right\}$$

This fixpoint for `r/1` is correct if `r/1` is called with a fully constrained argument — `?-r(b)` may fail, depending on whether `p(b, Y)` or `q(Y, b)` is first called. However if `r/1` is called with an unconstrained argument, then only one solution can be reached: $X = a$.

```
p(X)  :-  true ?  X = a.
p(X)  :-  true ?  X = b.

one_p(X)  :-  p(Y)  -> X = Y.
```

$$P_1 = \{\texttt{p/1}\}$$

$$\Phi_{P_1} \uparrow 0 = \{p(X) \leftrightarrow \bot\}$$
$$\Phi_{P_1} \uparrow 1 = \{p(X) \leftrightarrow (X = a \vee X = b)\}$$
$$\Phi_{P_1} \uparrow \omega = \Phi_{P_1} \uparrow 1$$

$$P_2 = \{\texttt{one\_p/1}\}$$

$$\Phi_{P_2} \uparrow 0 = \Phi_{P_1} \uparrow \omega \cup \{one\_p(X) \leftrightarrow \bot\}$$
$$\Phi_{P_2} \uparrow 1 = \Phi_{P_1} \uparrow \omega \cup \{one\_p(X) \leftrightarrow (X = a \vee X = b)\}$$
$$\Phi_{P_2} \uparrow \omega = \Phi_{P_2} \uparrow 1$$

$$\textbf{and}\ (\texttt{one\_p(X)};)_{\{X\}}$$
$$\Downarrow^*$$
$$\textbf{and}\left(\textbf{choice}\left(\ \textbf{and}\left(\textbf{choice}\left(\ \begin{array}{l}\textbf{and}\ (;)\ \texttt{?}\ \ \texttt{Y = a}\\ \textbf{and}\ (;)\ \texttt{?}\ \ \texttt{Y = b}\end{array}\ \right);\right)_{\{Y\}}\ \texttt{-> X = Y}\ \right);\right)_{\{X\}}$$
$$\Downarrow^*$$
$$\textbf{and}\left(\textbf{choice}\left(\ \begin{array}{l}\textbf{and}\ (;Y=a)_{\{Y\}}\ \texttt{-> X = Y}\\ \textbf{and}\ (;Y=b)_{\{Y\}}\ \texttt{-> X = Y}\end{array}\ \right);\right)_{\{X\}}$$
$$\Downarrow^*$$
$$\textbf{and}\left(\textbf{choice}\left(\ \textbf{and}\ (;Y=a)_{\{Y\}}\ \texttt{-> X = Y}\ \right);\right)_{\{X\}}$$
$$\Downarrow^*$$
$$\textbf{and}\ (;X=a)_{\{X\}}$$

Figure 4.5: A Non-Indifferent Program

4.5.2 Non-Indifferent Programs

Figure 4.5 shows a program which is not indifferent, along with its fixpoint and the actual AKL computation which will occur.

The AKL computation implies a fixpoint of

$$\Phi_{P_2} \uparrow \omega = \{p(X) \leftrightarrow (X = a \vee X = b), one\_p(X) \leftrightarrow X = a\}$$

which contains less possible successes than the fixpoint computed by the Φ_P operator. Since non-indifferent programs contain alternate solutions within their guards, some of which will be pruned away by the conditional guard operator, non-indifferent programs generally succeed less times than the fixpoint would imply. Non-indifference in conditional guards contains some similarities to red cuts, so this behaviour is not unexpected.

4.5.3 Non-Guard Stratified Programs

Figure 4.6 shows a non-guard stratified program, along with its fixpoint derived from Φ_P and an example computation. Not being guard stratified does not prevent a program from having its fixpoint calculated. However non guard-stratified programs contain hidden infinite loops which are caused by new environments contained in the guards being able to speculatively bind unconstrained arguments and continue. If the query `?-e(s(s(0)))` was tried, the program would terminate.

Guard stratification is not strictly needed to make negation monotone, as is the case with stratification of negation using the T_P operator. Predicates within guards are not assumed to be false, but rather $\bot$ until they are completely computed. As an example,

```
p  :-  not q.

q  :-  not p.
```

simply has a fixpoint of $\{p \leftrightarrow \bot, q \leftrightarrow \bot\}$.

```
e(0).
e(s(X))  :-  o(X)  ?  true.
o(s(X))  :-  e(X)  ?  true.
```

$$P_1 = \{\texttt{e/1}, \texttt{o/1}\}$$

$$\Phi_{P_1} \uparrow 0 = \{e(X) \leftrightarrow \bot, o(X) \leftrightarrow \bot\}$$

$$\Phi_{P_1} \uparrow 1 = \left\{ \begin{array}{c} e(X) \leftrightarrow (X = 0 \vee (X = s(X1) \wedge \bot)), \\ o(X) \leftrightarrow (X = s(X1) \wedge \bot) \end{array} \right\}$$

$$\Phi_{P_1} \uparrow 2 = \left\{ \begin{array}{c} e(X) \leftrightarrow (X = 0 \vee (X = s(s(X1)) \wedge \bot)), \\ o(X) \leftrightarrow (X = s(0) \vee (X = s(s(X1)) \wedge \bot) \end{array} \right\}$$

$$\Phi_{P_1} \uparrow 3 = \left\{ \begin{array}{c} e(X) \leftrightarrow (X = 0 \vee X = s(s(0) \vee (X = s(s(s(X1))) \wedge \bot)), \\ o(X) \leftrightarrow (X = s(0) \vee (X = s(s(s(0))) \vee (X = s(s(X1)) \wedge \bot) \end{array} \right\}$$

$$\vdots$$

$$\textbf{and}\,(\texttt{e(X)};)_{\{X\}}$$

$$\Downarrow^*$$

$$\textbf{and}\left(\textbf{choice}\left(\begin{array}{c}\textbf{and}\,(;\, X = 0)\ \texttt{?}\ \ \texttt{true}, \\ \textbf{and}\,(\texttt{o(X1)};\, X = s(X1))_{\{X1\}}\ \texttt{?}\ \ \texttt{true},\end{array}\right);\right)_{\{X\}}$$

$$\Downarrow^*$$

$$\textbf{and}\left(\textbf{choice}\left(\begin{array}{c}\textbf{and}\left(\textbf{choice}\left(\begin{array}{c}\textbf{and}\,(;\, X = 0)\ \texttt{?}\ \ \texttt{true}, \\ \textbf{and}\,(\texttt{o(X2)};\, X1 = s(s(X2)))_{\{X2\}}\ \texttt{?}\ \ \texttt{true},\end{array}\right);\, X = s(X1)\right)_{\{X1\}}\ \texttt{?}\ \ \texttt{true},\end{array}\right);\right)_{\{X\}}$$

$$\vdots$$

Figure 4.6: A Non-Guard Stratified Program

4.6 Related Work

The **FOUR** based logic used above has a strong relationship to the stable model semantics developed by Gelfond and Lifschitz [GL88]. The stable model semantics uses two sets: one for successes (S) and one for failures (F). An immediate consequences operator is developed that uses both S and F to construct further atoms; $H \leftarrow S_1, \ldots, S_n, \neg F_1, \ldots, \neg F_m$ results in H being included in the next S if each $S_i \in S$ and each $F_j \in F$.

The relationship between **FOUR** and the stable model semantics has been examined by Giordano et al [GMS96]: $G \leftrightarrow \bot$ if $G \notin S \cup F$, $G \leftrightarrow \top$ if $G \in S \cap F$, $G \leftrightarrow$ true if $G \in S - F$ and $G \leftrightarrow$ false if $G \in F - S$.

Naish [Nai89] uses four different sets to represent potential success and failure in committed choice programs, splitting success sets into definite and potential success sets and failure into definite and potential failure sets.

Gabbrielli and Levi [GL92] use an unfolding mechanism to derive a fixpoint semantics for committed choice programs. The unfolding mechanism is careful to retain the reactive behaviour of the program, ensuring that spurious successes are not possible and that deadlocks are correctly modelled.

The main advantage to using **FOUR** for formulating the semantics of a logic program is that the program retains its original logical flavour.

Chapter 5

The DAM

This chapter presents an abstract machine for use with the AKL: Doug's Abstract Machine, or the DAM. The DAM is intended for parallel execution of AKL programs and, therefore, has a number of features intended to model parallel execution. As a preliminary step, two existing abstract machines are described, to provide a foundation for the description of the DAM.

5.1 An Overview of Abstract Machines

Most programming languages have a computation model that can be reduced to a handful of basic operations. For example, Prolog can be reduced, via SLD resolution, to unification, term expansion and branch choice operations. C can be reduced to arithmetic, control flow and stack operations.

From this perspective, a programming language can be considered to have an ideal machine, which directly models the computation model of the language. The primitive operations for the language can then be encoded into a sequence of simple, assembly-like instructions. This "abstract machine" is similar in conception to an ordinary computer architecture, but may allow instructions that are not directly realisable on a real architecture; an example is the normal implementation of unification in Prolog, which may not terminate.

The instruction set, registers, data areas, etc. of an abstract machine can be optimised towards the computation model of a particular language. The language can then be compiled into these abstract machine instructions and executed by an emulator[1]. The advantages of using an abstract machine and compiler rather than an interpreter are obvious: rather than interpreting a program, the emulator does not need to do a great deal of work in decoding the program; each instruction tends to be fairly simple and can therefore be optimised effectively by the emulator; the compiler can be used to statically make optimisation decisions on the abstract instruction set.

This section briefly describes two abstract machines: the WAM and the JAM. Both these machines have influenced the design of the DAM and provide a basis for comparison. The JAM is a parallel implementation of Parlog. The WAM is included as an example of a reasonably efficient abstract machine design for a logic programming language (Prolog).

5.1.1 The WAM

The WAM, or Warren Abstract Machine, is an abstract machine formalised for use with Prolog. Elements from the WAM design can be seen in most abstract machines for logic programming languages. A complete description of the WAM can be found in [War83]. A more approachable description can be found in [AK91a].

The basic WAM architecture consists of three main data areas, a set of registers and an instruction set. These elements are summarised in table 5.1. The original WAM specification drew a distinction between

[1] Or an actual physical machine, if such a thing is achievable. An example is the CARMEL series of processors for FCP [HG91].

Data Areas	**Heap**	Stores permanent structures
	Stack	Stores temporary structures: environments and choice-points
	Trail	Used to reset some variables that have been bound on backtracking
	Push-Down List	A stack for use during unification
Registers	**P**	Program pointer
	CP	Continuation pointer (return address)
	E	Environment (stack frame)
	B	Choice-point
	A	Top of stack
	H	Top of heap
	TR	Top of trail
	HB	Heap position corresponding to creation of **B**
	S	Structure pointer into heap
	X0, X1, ...	Temporary registers
Instructions	get_constant, ...	Get instructions
	put_constant, ...	Put instructions
	unify_constant, ...	Unify instructions
	call, ...	Predicate call
	allocate, ...	Environment allocation
	try, ...	Choice point creation
	switch_on_term, ...	Indexing

Table 5.1: Elements of the WAM

the set of temporary registers, **X1, X2, ...**, and the set of argument registers, **A1, A2, ...**. In actual practise, no distinction needs to be drawn between them. (However, see section 5.1.2.)

Execution Model

The WAM treats a Prolog program, suitably encoded into WAM instructions in a similar manner to an ordinary imperative-language program. Each predicate call is treated in a similar manner to an imperative language procedure call; arguments are loaded into registers rather than the stack (an advantage when using an abstract machine is that you can use a huge number of registers) and the return address is loaded into the **CP** register rather than stacked. Argument values and the return address are only stacked when necessary. Each clause is treated in a similar manner to a procedure definition, with a stack frame (called the environment) being used to store local values between calls within the procedure.

The major point of departure from the traditional imperative execution model is the creation of a stack of choice points and a trail. Choice points record snapshots of the WAM state at the start of a predicate call. When the computation fails, usually as the result of a unification failure, the WAM state is unwound back to the first choice point on the stack, a process known as backtracking. This choice point can be used to recover the machine state and then try the next clause in the series of clauses defining a predicate. Upon execution of the last clause in a predicate, the choice point corresponding to that predicate is removed from the stack. A subsequent failure therefore, returns to the previous choice point.

To unwind the state of the WAM, a trail is used to record the changes that have been made since the last choice-point was created. When a variable is bound to a new value, the variable is added to the trail stack. Upon failure, the trail is unwound back to the position recorded in the current choice point. Each variable recorded in the trail is returned to the unbound state.

The WAM provides a primitive form of automatic memory management. New structures are built from blocks of word-size cells on the heap, the top of which is referenced by the **H** register. Upon failure, the **H** register is reset to the position given in the current choice point (and the **HB** register), recovering any memory used in speculative computation. The stack register, **A**, and the trail register **TR** are similarly reset.

Successful computations continuously increase the size of the heap and some form of garbage collection exists in most fully implemented WAM systems.

Representation

The most common representation of structures in the WAM consists of a series of cells. The cells are usually a convenient word size for the underlying hardware to implement (eg. 32 or 64 bits).

Prolog terms are represented by words containing a value and a tag. The tags take between 2 and 8 bits and give the type of the term. The remainder of the word usually contains a reference to another term or structure or, in the case of a constant, the value of the constant. Since all references in the WAM are word-aligned and the value of a term is often a reference to another term, the lower bits of the word are usually used to record the tag for references. The tags usually represent variables, constants, integers, references to structures and references to lists.

Bound and unbound variables share the same tag. Bound variables are represented by containing a reference to the term to which the variable is bound or, in the case of a constant, the actual value of the term representation. Unbound variables are represented by a reference to themselves. Since variables are words, they can be conveniently manipulated by placing them in registers; taking a copy of the value of an unbound variable turns it into a reference to that variable. A variable is bound to another term by placing the term representation into the variable's cell.

Complex structures with an arity greater than 0 are constructed on the heap. The structure consists of a header cell, containing the structure and its arity, followed by n cells containing the arguments of the structure. The entire structure is represented by a tagged word containing a reference to the structure. Since lists are used a great deal in Prolog, lists are simply represented by two cells. A reference to a list is given a different tag to that of a normal structure.

Instructions

Instructions in the WAM are represented by a series of byte codes, followed by arguments and possible immediate operands. The instructions can be divided into a number of families.

Get instructions test a register against some term, and if the register is an unbound variable, binds the variable to that term. If the register fails the test, then the computation fails and backtracking occurs.

Put instructions construct new terms. Put instructions are used to construct the terms used in a predicate call.

Unify instructions are used to test or construct the arguments of structures. A unify instruction is preceded by a get or put instruction, which puts the WAM into two possible modes: read or write. In read mode, the unify instructions act in a similar manner to get instructions. In write mode the unify instructions act like put instructions. When a get_structure instruction is executed, the WAM is placed in read mode if the argument being examined is bound to a structure, or write mode if the argument is an unbound variable.

Call instructions invoke predicates. Before calling a predicate $name/n$, the first n **X** registers are loaded with the arguments to the predicate. The **CP** register is set to point to the instruction immediately following the call instruction and **P** is set to the address of the code for the predicate being called. At the end of each clause, a proceed instruction returns by setting the **P** register back to the value of the **CP** register.

Environments are allocated by use of an *allocate* instruction, and deallocated by a *deallocate* instruction. Environments save the current **CP** register and provide a stack frame with a number of permanent registers, for use if terms need to be stored between predicate calls. The permanent registers take the place of the local variables of an imperative language and are referred to as **Y** registers.

Choice point instructions control the creation of choice-points. The set of clauses in a predicate is encoded into a try–retry–trust sequence. Try instructions create and initialise a choice point, and try the first clause in the sequence of possibly matching clauses. Retry instructions follow try instructions and use and update the choice point created by the first try instruction. Trust instructions remove the choice point and commit the computation to the last clause (since all previous clauses have been tried).

Indexing instructions allow the WAM to restrict the number of clauses tried. An argument to the predicate is tested, and if it is bound to a non-variable term, the set of clauses tried can usually be restricted. The switch_on_term instruction divides the predicate into four classes of clauses, based on the term in the

```
p([]).
p([a(X) | Rest]) :- q(f(X), X), p(Rest).

        switch_on_term Lv,Lc,Ll,Ls
Ls:     fail                    % No non-list structures
Lv:     try_me_else Lv1         % Try first clause
Lc:     get_nil X1              % Test against []
        proceed                 % Return if OK
Lv1:    trust_me                % Remove choice-point
Ll:     allocate 2              % Create environment
        get_list X1             % Test for list
        unify_variable X3       % Get a(X) part for later
        unify_variable Y1       % Save Rest
        get_structure X3,a/1    % Test for a(X)
        unify_variable Y2       % Save X
        put_structure f/1,X1    % Construct arguments to q
        unify_value Y2          % Make f(X)
        put_value Y2,X2
        call q/2,1              % Make the call
        put_value Y1,X1         % Get Rest back again
        deallocate              % Recover environment
        execute p/1             % Tail recursive call
```

Figure 5.1: Sample WAM Code

argument: a variable, a constant, a list or a structure. There are similar instructions for testing the actual values of arguments.

A sample piece of Prolog code and equivalent WAM instructions is shown in figure 5.1

Optimisation

There are several simple optimisations that are usually performed on WAM code:

- The last call in any clause can usually have the clause environment removed before it is invoked and the continuation point for the clause used for the call. This optimisation is a generalisation of the tail-recursion optimisation.
- If a predicate has only one applicable clause, either via indexing or from only having one clause, then no choice point needs to be created.
- Variables are always bound so that the younger variable points to the older variable. The stack is always placed so that stack variables always refer to the heap. With this assumption, only those variables on the heap outside the range between the **HB** and **H** registers need to be trailed.
- New variables could be created on the heap. However many variables are created as return arguments to calls within a clause. These variables can be created as unsafe variables within the environment of the clause. If an unsafe variable is still unbound upon its last use, the variable is moved to the heap.

5.1.2 The JAM

The JAM, or Jim's Abstract Machine, was developed by Crammond to provide an abstract machine for the parallel execution of Parlog. A complete description of the JAM can be found in [Cra88]. The JAM shares several common points with the WAM, but has been designed to accommodate the attributes of Parlog: parallelism, suspension and guard computations. The instruction set and unification primitives are almost

identical to the WAM and will not be discussed unless some significant difference occurs. See section 5.1.1 for a discussion of the WAM.

The JAM, like Parlog, is similar to Conery's And/Or process model [Con83]. Each conjunction of goals is represented by an And-process. Each uncommitted predicate call is represented by an Or-Process. A tree of And- and Or-processes is used to represent the computation. Most of these processes will not actually be running at a given time. They will either be suspended and waiting for a variable to be bound, or queued and waiting for a processor.

Each actual processor in the JAM represents an independent element, capable of executing any of the processes in the And/Or tree.

Execution Model

The execution model of the JAM is that of a set of processors, each with a work queue containing runnable processes. A processor with no work on its queue can search other processors for work and request work from another processor's queue. Each process represents a node in the And/Or tree.

The memory architecture of the JAM is more complex than that of the WAM, although no trail is necessary as backtracking does not occur. The following data areas are used by the JAM (running from high memory to low memory addresses):

Run Queue The list of runnable processes.

Process Stack Temporary environment space for each process.

Temporary Heap A heap for use while executing a guard.

Permanent Heap The heap for constructing terms.

Argument Stack Sets of arguments for each process.

Program Program area.

The temporary and permanent heaps may be merged. Data areas from each processor are interleaved, so that all stacks, heaps and queues are grouped together; the JAM uses a similar system to the WAM in binding high address variables to low address variables.

An attempted unification in the JAM can result in three possible states: success, failure or suspension. Suspension occurs when a variable external to a guard is tested against a term and that variable is unbound. At this point, the process is added to the variable's suspension list and the process is suspended. When that variable is bound the process is woken, and the test re-applied.

A call in the JAM can either proceed sequentially or in parallel. Sequential calls can be executed directly by the processor making the call. Parallel calls are queued, to be executed when the processor has no more work to do. In both cases, an area of the argument stack is allocated and arguments pushed onto the stack. A suitable Or-process is created to execute the resulting call.

Clauses in the JAM can also be tried either in sequence or in parallel, using a modification of the try–retry–trust sequence of the WAM.

Representation

Terms in the JAM are represented by a similar tagged format to those of the WAM. Unbound variables are represented by a separate tag to references, since an unbound variable may have a list of suspended processes attached to it.

Processes are represented by a process structure, which is built on the heap for each process created, and recovered or garbage collected at the end of the process. Each process contains a reference to its parent process, a program pointer, a count of the number of child processes, a flag word, a pointer to the root process, a pointer to the process arguments, a pointer to the process environment and a link for the process's attachment to either a suspension list or processor queue.

There is no need for a separate choice point structure for the JAM, since an or-process structure serves a similar purpose.

```
mode p(?).

p([]).
p([X | R]) <- : q(X), p(R).

        try L1                    % Parallel clause try
        try_one L2                % Final clause
L1:     wait_nil,A1               % Wait on first argument for []
        commit                    % Commit to this clause
        proceed                   % End of clause
L2:     wait_list,A1              % Wait on first argument
        allocate 2                % Keep X and R
        unify_y_variable Y1       % Get X
        unify_y_variable Y2       % Get R
        commit                    % Commit to this clause
        push_y_value Y1           % Set up q(X)
        call_promoted q/1,1       % Call q(X) in parallel
        put_y_value Y2,A1         % Call p(X)
        call_promoted_last p/1,1  % Tail recursive call
```

Figure 5.2: Sample JAM Code

Instructions

The instruction format for the JAM is similar to that of the WAM. Get, put, unify and indexing instructions play an almost identical role to those of the WAM. The JAM adds instructions to allow guarded computations, wait instructions and sequential or parallel call and try instructions.

The *commit* instruction forces all other branches of an or-process to be killed, leading to a single clause being available for promotion.

Wait or suspension instructions force a suspension of a process if the variable that they are testing is unbound. Otherwise, wait instructions behave in a similar manner to get instructions.

An example piece of Parlog code and equivalent JAM instructions are shown in figure 5.2.

5.2 Underlying Architecture

The machine architecture underlying an abstract machine has a significant effect on the design of an abstract machine. In theory, an abstract machine can be implemented on any architecture, but for efficiency reasons it is best to take advantage of the strengths of a given machine and avoid its weaknesses.

There is no unifying paradigm for parallel architectures similar to the familiar von-Neumann architecture for sequential machines. Machines are usually classified by their approach towards processor communication, synchronisation, memory locality and the expected number of processors running in parallel. All of these four factors affect each other, so some combinations are more likely than others.

All parallel systems necessarily involve a set of processors executing instructions semi-independently. Parallel machines are usually categorised by the Flynn taxonomy [Fly66]. The first rough division between parallel machines is between single instruction, multiple data (SIMD) machines and multiple instruction, multiple data (MIMD) machines. The processors of SIMD machines all execute a single program in lockstep. SIMD machines are well suited to problems that exhibit a high degree of data parallelism, such as database queries or matrix calculations. SIMD machines lend themselves well to massive parallelism, since there are few control problems associated with them. In MIMD machines, each processor can execute an individual program. MIMD machines are suited to most general forms of parallelism.

Architectures can be roughly divided into scalable architectures (where the performance of the system is roughly proportional to the number of processors present) and non-scalable architectures (where performance of the system tends to tail off as new processors are added). Another way of looking at these

architectures is by describing them as massively or modestly parallel. While not a hard and fast distinction, massively parallel architectures tend to have over 100 processors, modestly parallel architectures below that.

Parallel systems can be divided into fragmented memory and unfragmented memory architectures. Systems with unfragmented memory share a common memory pool. Contention tends to limit unfragmented memory systems to a modest level of parallelism. Each processor can access any area of unfragmented memory by address. Fragmented memory systems break the memory space into independent units, associated with some controller, such as a processor or a dedicated memory controller. Access to fragmented memory is mediated by the controller.

Usually processors need to communicate with other processors. Shared memory communication systems allocate an area of memory common to both processors. Information can be deposited and retrieved from an agreed rendezvous point in memory. Shared memory communication systems usually require a system of locks to prevent contention. Message passing communication systems associate a message buffer with each processor. Any communication can be represented by sending and processing messages.

Some real-world examples of this taxonomy are: the first Connection Machine (scalable, SIMD, fragmented memory, message passing), the SGI (non-scalable, MIMD, unfragmented memory, shared memory) and the Transputer (scalable, MIMD, fragmented memory, message passing).

Each type of choice described above represents a trade-off between competing sets of requirements. Contention tends to force massively parallel systems to use fragmented memories. Fragmented memories tend to enforce a message passing communication system. Some hybrid systems impose a hierarchy of organisation to support the efficiency of shared-memory communication between small groups of processors while retaining the massive parallelism possible through local memory and message passing. An example is the Parallel Inference Machine [Got87], where small clusters of processors share local memory but messages are exchanged between clusters. The KSR1 [FBR93] maintains a logically unfragmented address space, with each processor maintaining a cache and no processor "owning" an address. The Data Diffusion Machine (DDM) [WH88] allows shared memory access to a system with local memory by providing a hierarchy of data directories. In both the KSR1 and DDM, a memory request can be referred to another processor and data copied to the requesting processor; a memory address does not correspond to a fixed memory location and multiple copies of data for the same memory address may exist.

5.2.1 Target Architecture

In designing an abstract machine for the AKL, a target architecture needs to be chosen. The underlying architecture of the system will have a great influence on the final design of the abstract machine.

The choice of target architecture for the DAM has been partially influenced by the availability of hardware: no SIMD, scalability or massive parallelism. The machines available all use global, shared memory for parallelism. However local or global memory can easily be simulated on different architectures, as can shared memory or message passing. The choice of a target architecture in these cases rests more with the nature of the task.

Logic programs tend to share structures a great deal. Large structures such as lists can be referenced by a single pointer to the start of the list. A strictly distributed structure would require copies of terms to be made when a non-local processor needs to refer to those terms. A variety of incremental copying schemes have been designed for use with distributed architectures [Nak92, Foo94]. The existence of the DDM suggests that shared memory could be used for reading terms, leaving the actual location of the term to the underlying system [RDC92].

The issue of using shared memory with locks or message passing for updating shared structures is more problematic. In an implementation of AKL, most data structures will be updated by a single processor most of the time. If it could be arranged that the structure representing data is located on the processor that "owns" it, then a message passing system would allow that processor to update the state of the box without the need to lock the data structure each time. The message passing approach also encourages an object-oriented view of the AKL system, with each piece of data acting as an object. That advantage may be offset by the data structure now having a preferred processor, leading to a granularity problem. In particular, stream and-parallelism in the AKL tends to result in several processes sharing an and-box; each non-owning worker would need to request changes from the owning worker, leading to a bottleneck. An initial design

did take the message passing approach, with disappointing results. LeBlanc and Markatos report that the advantages to message passing are often outweighed by the effects of load imbalance [LM92].

The assumed target architecture for the abstract machine can therefore be summarised as:

- *Multiple Instruction, Multiple Data.* Each processor can execute a separate thread of the computation.
- *Modestly Parallel.* Only a few (possibly less than 10) processors. Since logic programs can readily create parallel nodes for both and- and or- nodes, it can be assumed that the processors rapidly become saturated with work and that a great deal of code is going to be called sequentially.
- *Shared Memory.* All terms and structures are readily available by reference, with no copying needed between processors.
- *Locking.* Access to shared structures is via explicit locking, rather than message passing. Each structure has no preferred processor.

5.2.2 Locking

Each structure in an AKL computation may need to be locked, and may stay locked for an indeterminate amount of time (e.g. while localising a variable, section 5.3.1). In systems with hardware locks, only a limited number are available. In systems which provide operating system based software locks, the locks tend to be fairly large (eg. 20 bytes for a Solaris 2.3 lock) and their use needs to be limited.

An alternative is to directly use a native atomic update instruction. Use of such hardware specifics, however, tends to destroy the portability of the system.

The approach taken here, and used in [Cra88], is to use a limited number of hardware/software locks, in conjunction with hashing. Each structure that needs to be locked has an associated status bit assigned to it. Since each structure can be identified by a reference to it, the structure is locked by first locking a hardware/software lock, based on a hash-value derived from the reference. When the lock is acquired, the locking bit in the structure can be set and the hardware/software lock unlocked. A structure can be unlocked simply by clearing the locking bit in the structure.

A test of the hashing technique, run over a million operations and 12K words of memory on an SGI system with 6 processors produces the results summarised in table 5.2. Three tests were run: using hardware locks only, using mixed hardware/memory locks, with each processor ranging over all memory and using mixed hardware/memory locks, with each processor restricted to a separate area of memory. The times represent a lock followed by an immediate unlock.

In general the use of hardware/memory or software/memory locks imposes a 30% – 40% performance penalty over simply using hardware or software locks. With the exception of 4 locks, performance is relatively insensitive to the number of locks available for all tests. An increase in the number of processors increases the probability of contention, leading to a consistent increase in times, especially in the case of 4 locks, where contention is greatest. Memory locking over separate areas shows a consistent performance improvement over the shared memory tests. This improvement cannot be due to a lack of contention, since the hardware locks are still shared between the memory areas. The costs of ensuring cache coherence seems to be a likely explanation.

An alternative to using hardware locks is to use a software locking scheme such as that of Michael and Scott [MS93]. Software locks can show significant performance improvement over hardware locks. However special requirements, such as speed bounds, makes implementation of software locks unattractive.

The locking model chosen for the DAM is a mixed hardware and memory or software and memory locking scheme, depending on hardware and operating system support, with more locks than processors.

5.2.3 Memory Allocation

Since the expected target architecture for the DAM follows a global memory model, it is possible to allocate all memory requirements for each processor from a common pool. While this is memory efficient, it requires locking for each request for memory. An alternative is to allow each processor access to a private heap, eliminating the need for locking.

		Locks				
Processors	Model	4	8	16	32	64
1	Lock	3.0	3.1	3.0	2.9	3.3
	Memory	4.0	3.9	3.9	3.8	4.5
	Separate	3.8	3.8	3.9	3.9	4.5
2	Lock	3.7	3.1	3.9	3.8	3.9
	Memory	5.1	5.3	4.8	4.9	5.4
	Separate	4.1	4.5	3.9	4.8	4.7
3	Lock	3.8	3.9	3.7	4.0	3.5
	Memory	5.4	5.1	5.6	5.4	4.8
	Separate	5.3	4.5	5.0	5.0	4.6
4	Lock	4.1	3.9	4.1	4.0	3.8
	Memory	6.0	5.8	5.3	5.5	5.5
	Separate	5.5	5.0	5.0	5.4	5.2
5	Lock	4.5	4.4	4.2	4.1	4.0
	Memory	6.1	5.9	6.0	5.8	5.6
	Separate	5.5	5.4	5.7	5.3	5.1
6	Lock	5.1	4.5	4.3	4.2	4.2
	Memory	6.4	6.1	6.0	5.7	5.8
	Separate	5.8	5.7	5.6	5.6	5.4
All results in s, $\pm 20\%$						
Lock: Hardware locking only						
Memory: Hware./memory locking, shared memory						
Separate: Hware./memory locking, separate memory						

Table 5.2: Comparison of Direct Hardware Locking and Hardware/Memory Locking

	Processors					
	1	2	3	4	5	6
Overhead	0.2	0.2	0.2	0.2	0.2	0.2
Separate	1.0	1.0	1.1	1.1	1.1	1.1
Locking	6.3	18.8	29.1	35.9	43.6	44.9
All result in s, $\pm 10\%$.						
Overhead: test overhead with no allocation						
Locking: single heap with locking						
Separate: independent heaps						

Table 5.3: Comparison of Heap Allocation Strategies

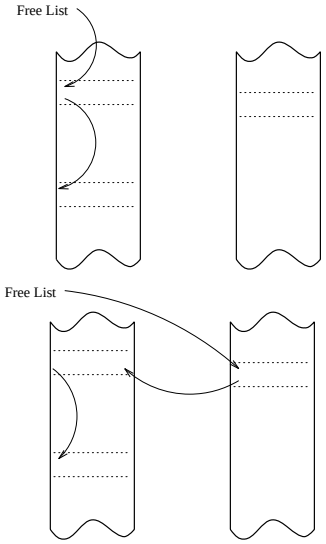

Figure 5.3: Memory Deallocation Across Processors

Intuitively, a single heap allocation strategy represents a bottleneck. As a test, a million allocation/deallocation pairs of randomly generated block sizes were run on a simple top-of-heap allocation strategy for 1–6 processors, performing either with a single heap and locking, or each with a private heap. The results are summarised in table 5.3. Using locking results in a massive overhead on what should be a simple operation. Each processor therefore should be given a private heap to manipulate.

Heap allocation follows a "fast fit" allocation strategy [Sta80]. The heap consists of a top of heap pointer and a set of free lists, grouped into blocks of size ranging from 1 word to 64 words (the maximum structure size). A request to the heap for memory first checks to see if an appropriate block is already available in the free list. Otherwise, a block is allocated from the top of the heap. Deallocated blocks are added to the appropriate free block list.

A block of memory that has been allocated by one processor can be deallocated by another processor. If the block is returned to the allocating processor's heap, then there will be a need for locking as the block is deallocated — something to be avoided. Since the block to be deallocated is under the control of the processor deallocating the block, it makes sense to simply add the memory block to the free list in the heap of the deallocating processor (see figure 5.3).

A comparison of a simple top-of-heap allocation and the fast fit method was run using ten million randomly generated block sizes between 1 and 64 words. The results, correcting for overhead, were 4.3s for the top-of-heap allocation method, and 6.4s for the fast fit method. This represents an overhead of 48%, which is acceptable in exchange for greatly reduced memory usage.

5.3 Execution Model

The DAM builds an And-Or tree of and-, choice- and bagof-boxes (figure 5.4). Or-boxes have been eliminated by combining the nondeterminate promotion and guard distribution rules into a single operation.

Active boxes are boxes that have further work to do. *Waiting* boxes are boxes that are waiting for an external event (such as a child box to complete, a variable to be bound, etc.). *Killed* boxes are boxes that have either failed or been killed by a parent box.

Each box contains two program pointers. The *program pointer* of a box contains the next abstract instruction that the box will perform if the box becomes active. Each processor is assigned a machine capable of executing the work waiting to be done in a box. This machine is called a *worker*. The *continuation*

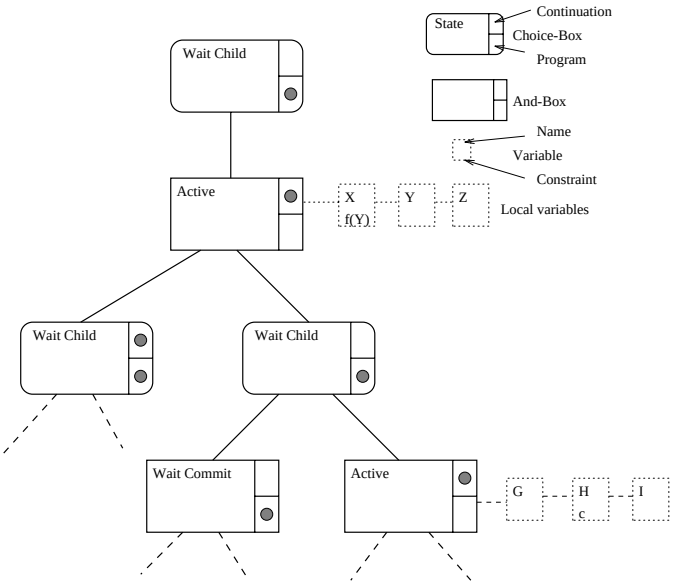

Figure 5.4: The DAM And-/Choice-Box Tree

pointer of a box contains the next abstract instruction that the parent box will perform if the current box suspends, fails or completes.

And-boxes contain links to a number of other objects. The local variable list is a linked list of all those variables local to the and-box. The localised variable list contains local copies of variables obtained from a parent and-box (see section 5.3.1). An and-box is *quiet* if there are no localised variables within the box that have not been constrained by their parent variables.

Each box also contains a count of the number of workers currently active within it, the number of child boxes that it has, and the number of suspended child boxes. A box is *suspended* if it is in a waiting state, there are no workers active within it, and the number of child boxes equals the number of suspended child boxes. An and-box is *stable* if the box is suspended and quiet.

5.3.1 Constraints

The constraint system used in the DAM is the familiar constraint system of equality over Herbrand terms [Llo84]. Terms are built from constants, function symbols with a fixed number of arguments, and variables.

Variable Localisation

The AKL computation model allows a hierarchy of constraints, with both variables and constraints being *local* to a specific and-box. A child and-box inherits the constraints and variables of its parents, but may add additional variables and constraints itself. The scope of a variable or constraint, therefore, encompasses the and-box that it is local to, and any child boxes from that and-box. If an and-box adds a constraint to a variable local to some parent and-box, some mechanism is needed to ensure that the scope of the constraint is respected.

One solution to this problem is to use hash-tables to record the local constraints on a variable. This approach has been taken by Moolenaar and Demoen for the ParAKL system [MD93]. Another approach is taken by Montelius and Ali in the Penny parallel abstract machine, where each variable that is constrained further down the and-box hierarchy maintains a list of suspensions [MA96]. Both these approaches have a non-constant access time for determining bindings.

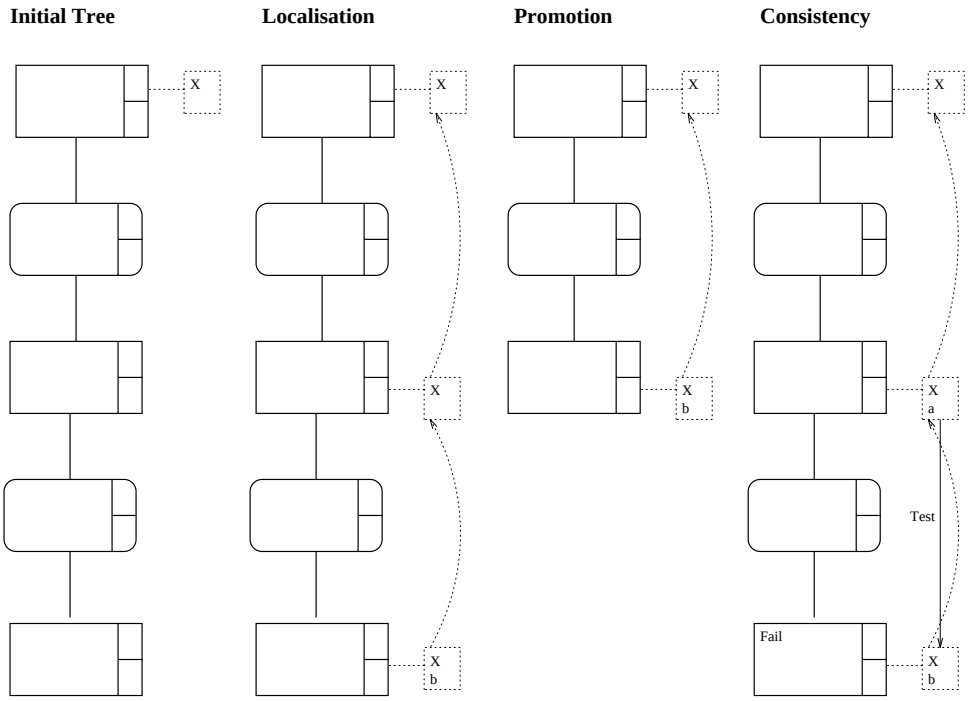

Figure 5.5: Variable Localisation in the DAM

The approach taken by the DAM is to treat each variable as a parent variable and a hierarchy of localised copies. When a variable not local to an and-box is constrained, a chain of localised variables is created between the parent and-box of the variable and the and-box which constrains the variable. In effect a new variable (with binding) is created that is valid for the constraining and-box. In most cases further references to the variable will use this localised copy, leading to a constant access time. This localised variable can then be used within the scope of the and-box as if it was the original variable referred to. Upon promotion, the immediate parent of the localised variable can be given the constraint contained in the localised variable.

If a parent variable is constrained the tree of localised variables below the parent can be tested, failing any and-boxes that contain incompatible constraints.

The operations of localisation, promotion and parent constraint testing are shown in figure 5.5.

Localised variables allow constant access time. However, creating a localised variable is a non-constant time operation, as each level of and-box needs to be searched for an existing localised variable and, if absent, a new localised variable created. Creating a localised variable also requires a variable to be allocated to each level as opposed to a single hash-window cell in ParAKL or suspension in Penny. In practise, AKL programs tend to have a fairly shallow nesting of and-boxes [Mon97], reducing the cost of long chains of localised variables.

Adding Local Variables

As local variables are created, they need to be added to the list of local variables maintained by the and-box that owns them. This list is maintained so that the variables in a promoted and-box can have their and-box entries adjusted to be that of the new and-box. The simplest way to avoid contention when adding to an and-box is to lock the and-box as each new variable is created.

There are two reasons for wanting to avoid locking the and-box on variable creation: variable creation is extremely common and having to lock every time a variable is created will lead to a loss of performance and producer-consumer type programs tend to have producers and consumers sharing an and-box, as and-boxes

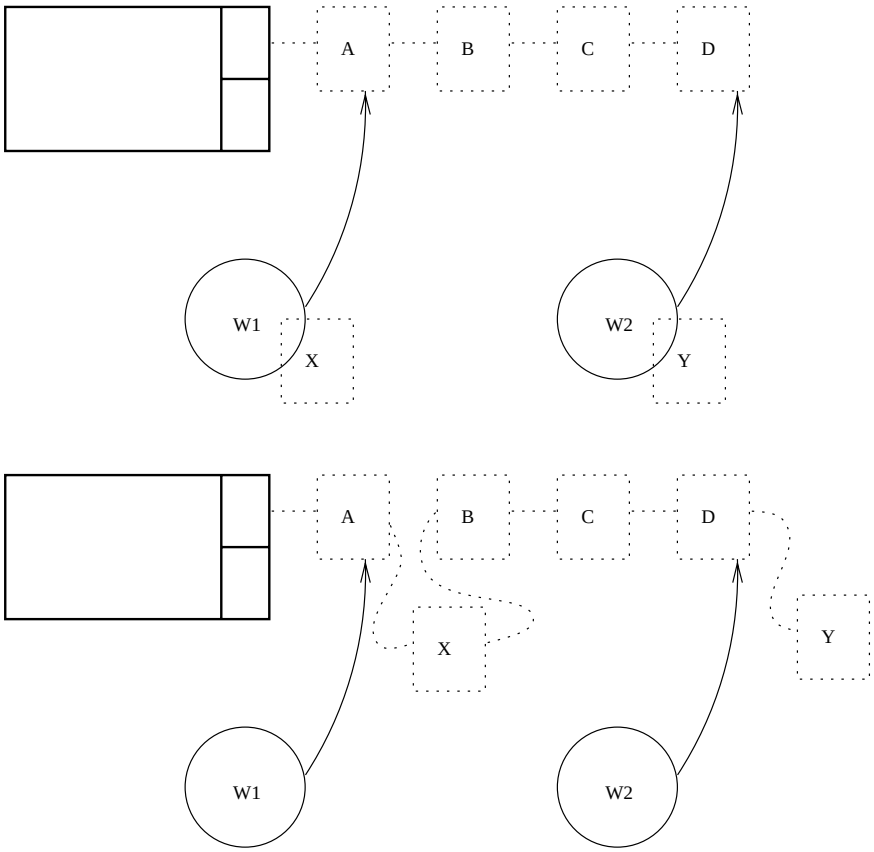

Figure 5.6: Adding a New Local Variable

are promoted, and the shared and-box becomes a bottleneck.

It is possible to avoid the expense of locking, in most cases, if each worker uses a local variable to act as the head of the local variable list. Since each worker is holding a unique local variable as the head of the local list, there will be no contention when adding a new variable. The new variable can then be inserted into the local variable list without locking. Figure 5.6 shows an example of two workers adding variables.

An alternative to maintaining a list of local variables is to mark a promoted and-box as promoted and to allow the locality checks to follow the chain of promoted and-boxes up to the actual and-box [MA96]. This method is superior to the method in the DAM, as it both saves on the memory used by the list and the time taken to promote the variables. The cost of following the and-box chain is relatively trivial.

5.3.2 Indexing

The standard WAM implementation allows indexing on the some argument to a call, usually the first although some implementations (eg. NU-Prolog) allow indexing on any argument. Three instructions allow different code branches to be selected, based on the overall type of the argument (variable, constant, list or structure) and the exact value of the constant or structure. In most cases this level of indexing is sufficient; an initial split between the sets of possible clauses is created, and each clause from the set tried in turn. Singleton clause sets can executed immediately without creating a choice-point.

In the DAM, the creation of choice- and and-boxes is a more expensive operation than the creation of a choice-point in the WAM, and it makes sense to pay more attention to indexing issues. If a single clause can be found, then a choice-box and and-box may not need to be created. Instead of simply indexing on an argument, a more thorough indexing on all arguments should uncover more cases of simple determinism.

Naively, it should be simple to generate a decision tree covering all possible combinations of bound and unbound arguments. However, the size of the decision tree rapidly grows and, in general, creating a decision tree is NP-Hard [PN91]. An example of a predicate that generates an exponential decision tree is `p/6`:

```
p(0, 1, 0, _, _, no).
p(0, _, _, 1, 0, no).
p(1, 0, _, 0, _, no).
p(_, 1, 0, _, 0, no).
p(_, _, _, _, _, yes).
```

`p/6` is a disguised form of the 3-satisfiability problem [GJ79] for the formula $\mathcal{F} = (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_4 \vee x_5) \wedge (\neg x_1 \vee x_2 \vee x_4) \wedge (\neg x_2 \vee x_3 \vee x_5)$. Any path in the decision tree that commits to the last clause can be read as a solution to the problem.

There are several ways of avoiding exponential decision trees. If the expected input modes are known (e.g. from a moded logic programming language such as Concurrent Prolog or Pandora) then the decision tree can be generated directly [KS88, KT91]. Alternately, partial indexing is possible, abandoning the indexing when the decision tree becomes too complex and simply trying clauses [SCWY91b].

The approach taken in the DAM is to use the *clause set* approach developed for NUA-Prolog [PN91]. The set of possible clauses is represented by a bit-vector, with each set bit representing a candidate clause. Arguments can then be examined and the set of candidate clauses intersected with the current clause set. If the clause set reduces to a single bit, then the corresponding clause can be committed to. If the clause set reduces to more than one bit, the clause set can be used as a filter to select clauses.

The clause set approach avoids NP-Hardness by never maintaining a complete path from argument bindings to clauses. Any attempt to find a solution to $\mathcal{F}$ would require intersecting all combinations of clause sets to see which reduce to the singleton last clause. The tradeoff is the run-time intersection of bit-vectors — reasonably fast on any halfway decent computer.

5.3.3 Waiting on Variables

Creating and destroying boxes for speculative computation can be extremely wasteful. In many cases, much speculative computation can be avoided if the full expansion of the box is left until some variables become bound. As an example, consider the predicate `stack/2`:

```
stack([], _) :- true ?  true.
stack([push(X) | R], S) :- true ?
       stack(R, [X | S]).
stack([pop(X) | R], S) :- true ?
       S = [X | S1],
       stack(R, S1).
stack([top(X) | R], S) :- true ?
       S = [X | _],
       stack(R, S).
```

If the first argument of a call to this predicate is sufficiently bound, the abstract machine can immediately choose a suitable clause without creating any speculative and-boxes. If the wait-guards in `stack/2` are replaced by conditional or commit guards, the first argument must be bound to ensure quietness.

In these cases it is sensible to delay the execution of a choice-box until the arguments to the choice-box are sufficiently instantiated. The DAM therefore allows boxes to be added to the dependents list of a variable and the box then waits on the variable. When a variable is bound and the dependents list checked, the box can be made active again.

Boxes with wait guards can benefit from waiting on some variables. However these boxes must eventually be made available for nondeterminate promotion. Two forms of waiting on variables are required. *Strong waiting* means that the variable must be instantiated before the choice-box will proceed. Strong waiting is suitable for conditional and commit guards. *Weak waiting* allows the box to be *forced*. A forced box ignores any further weak waiting on variables and proceeds to expand the choice box.

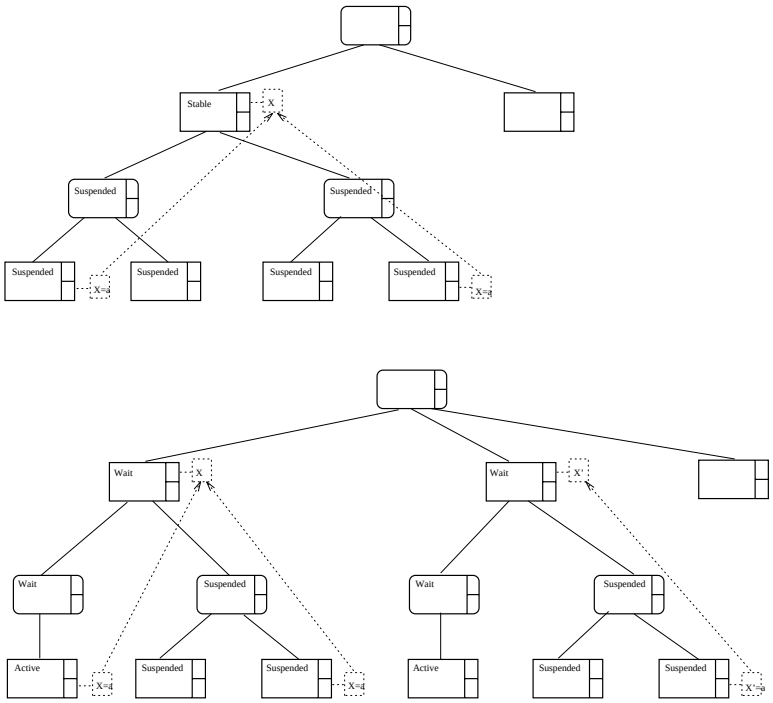

Figure 5.7: Nondeterminate Promotion in the DAM

5.3.4 Nondeterminate Promotion

Nondeterminate promotion is achieved in the DAM by copying part of the box tree generated by the program's execution. Copying is also used in Montelius and Ali's implementation of Penny [MA96].

When an and-box becomes stable, the tree below the and-box is searched for a choice-box which has as its first and-box an and-box which has completed its guard and is waiting for commitment. This choice-box can then be used as the basis for a single nondeterminate promotion. Only the first child box in a choice-box is examined; completed candidates further down the list of and-boxes are ignored to ensure correct behaviour within a box guarded by conditional guards.

Choice-boxes that are weakly waiting on a variable may also be used for nondeterminate promotion. A weakly waiting choice-box may be forced, leading to a complete expansion of the box and a possible further nondeterminate promotion.

The search for a suitable candidate for nondeterminate promotion proceeds depth-first, left-to-right. There is no particular reason for this choice, beyond the desire to emulate the Prolog search rule and the simplicity of the search. A more intelligent search rule would examine other alternatives and choose those with the minimum number of alternatives and the minimum number of constraints. Moolenaar and Demoen have found that an intelligent search rule can improve search performance by 2–3 times [MD94]. Another approach would be to provide a programmable search system, similar to Pandora [Bah91].

When a choice-box is selected for copying, a duplicate of the parent and-box and all child-boxes is created and added to the parent choice box. The and-box selected for promotion is moved to the choice-box and made active. In addition the choice-box from which the and-box was taken may become active, due to having only one child left and being eligible for determinate promotion.

When a duplicate of a configuration is made, any unbound local variables also need to be copied, as do any structures that are not ground.

An example copying operation is shown is figure 5.7. During nondeterminate promotion, once a choice-box has been selected for nondeterminate promotion, the and-box above it and the and-box's children, along any with variables and bindings within the scope of the tree, is duplicated. The duplicated box tree contains

Message	Description
Active	The box has become active.
ActiveChild	A child box has become active.
AddChild	Add a new child box.
Collect	Collect a term for a bagof-box.
Commit	Commit to determinate promotion.
Copy	Make a copy of the box and children.
CopyChild	Add a copied child.
DoneChild	A child box has been promoted.
Enter	A worker has entered this box
Exit	A worker has left this box
Fail	The box has failed.
FailChild	A child box has failed.
Force	The weakly waiting box has become active.
Kill	Kill the box.
NonDetCandidate	Request a candidate for nondeterminate promotion.
NonDetPromote	Nondeterminately promote this box.
Promote	Promote the variables local to this box.
Recover	Recover this box for the heap.
RecoverChild	A child box can be recovered.
RequestConditional	Request a conditional pruning.
RequestCommit	Request a commit pruning.
Suspend	The box has suspended.
SuspendChild	A child box has suspended.
TakeContinuation	Acquire the continuation pointer.

Table 5.4: Box Messages in the DAM

all boxes except for the child of the choice-box selected for nondeterminate promotion. As a result, there are now two configurations, with at least one configuration ready for determinate promotion.

5.3.5 Box Operations

Boxes communicate with each other by (conceptually) passing messages. Most messages work upwards from child boxes to parent boxes, however a parent box may kill a child box, or cause it to commit or promote. The messages that may be passed between boxes are shown in table 5.4 and the senders and receivers are summarised in figure 5.8. Boxes, usually via abstract machine instructions, may also send messages to themselves. In practise, messages sends are simply direct procedure calls. Multiple messages are handled by locking the box, ensuring that only one message is processed at a time.

Bagof-boxes use similar messages to choice-boxes, although bagof boxes cannot cause an and-box to commit. Bagof-boxes may also be sent the Collect message, adding a new element to the list generated by the bagof-box. Variables may also send Fail or Active messages to boxes.

When a box is created, it is added to the parent box by means of an AddChild message. If the box is to be run immediately, it also takes a worker from the parent box.

The box then continues until it either suspends, fails or promotes. Upon suspension, the box sends a SuspendChild message to its immediate parent, which can in turn suspend and send further SuspendChild messages. Suspended boxes can be woken by Active or NonDetPromote messages, whereupon they send ActiveChild messages to their parents. If the box fails, it sends a FailChild message to the parent. If the parent box is a choice-box, the choice-box records the failure and waits until only one child box remains, in which case the remaining child box is sent a Commit message.

When an and-box promotes, it sends a Promote message to the grandparent and-box to pass all local variables upwards. It also sends a TakeContinuation message to the parent choice-box, to acquire the stored continuation point in the choice-box, if any exists. Finally the and-box sends a DoneChild message upwards

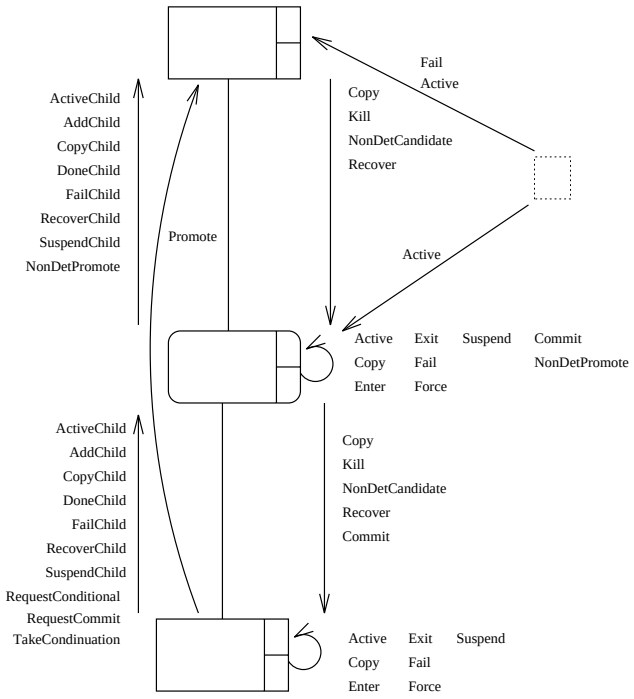

Figure 5.8: Box Messages Senders and Receivers

to detach the and-box and parent choice-box and deallocate them. All further processing is performed by the grandparent and-box, which also inherits the worker which performed the promotion.

If an and-box becomes stable, the and-box recursively sends NonDetCandidate messages to its children, searching for a suitable candidate choice-box with a first child available for nondeterminate promotion. The candidate is sent a NonDetPromote message, which in turn sends a NonDetPromote message to its parent and-box. The and-box then makes a copy of itself and all child-boxes via Copy messages. As boxes copy, they attach themselves to the appropriate parent box by means of CopyChild messages.

A box may be killed if a parent box has failed, or a pruning operation has eliminated the box. When a box receives a Kill message, it terminates, and sends further Kill messages to all its children.

When a box terminates, either through failure, promotion or being killed, it may be recovered for reuse by the heap. Promoted boxes may be recovered immediately. Failed and Killed boxes may still have workers active within them. As a result, they may not be immediately recoverable. When a worker finds itself in a dead box, it exits the box, and sends a RecoverChild message to its parent. The top-level dead box accumulates all RecoverChild messages until all workers have left that part of the tree, and then sends Recover messages to each child box. A box receiving a Recover message is deallocated.

5.4 Abstract Architecture

The abstract architecture of the DAM is based around a set of workers, with each worker assigned to a separate processor. Each worker maintains a heap, from which it can allocate memory for terms, variables, box structures, environments and other objects. Each worker also maintains a trail for use during copying. All workers share a lock table of hardware locks, program and symbol table stores, a common work queue and a global status register. Figure 5.9 shows the an outline of the DAM abstract architecture.

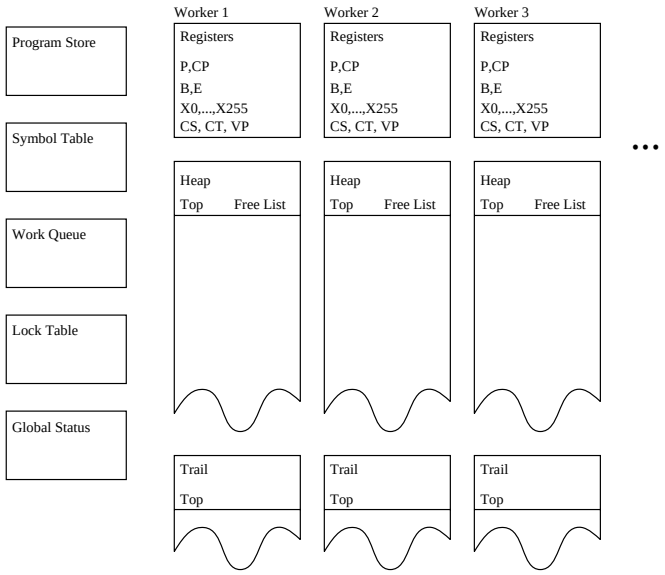

Figure 5.9: DAM Abstract Architecture

5.4.1 Registers

Each worker also maintains a register set, which is used to execute the program. The DAM registers consist of:

P The program counter: a pointer into the program store giving the next abstract instruction to execute.

CP The continuation pointer: a program address to continue with after the current box has suspended or ended.

B The current box: a pointer to the box environment that the program is executing in.

E The environment pointer: a pointer to an array of permanent registers, for use by the current box.

VP The local variable pointer: a pointer to the local variable to use when adding new local variables to the current and-box.

CS The clause set register: a 32-bit set for computing clause sets.

CT The clause table register: a pointer to a clause table corresponding to the bits in the clause set.

X0,...,X255 Temporary registers: A set of temporary registers, each holding a single term.

In addition to these registers, the permanent registers provided by the environment are denoted by **Y0, Y1, ...**.

5.4.2 Instruction Format

The instruction format for the DAM is a sequence of 4-byte words, the first byte containing the instruction code and the following three bytes containing simple arguments. Any immediate arguments, such as constant values or program addresses to jump to, follow the initial instruction word. All immediate arguments are word-sized.

Sample instructions and their layout are shown in figure 5.10. The actual instructions will be introduced in the sections describing the operations of the DAM.

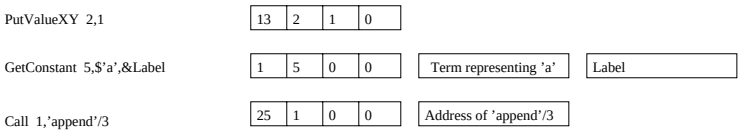

Figure 5.10: Instruction Formats for the DAM

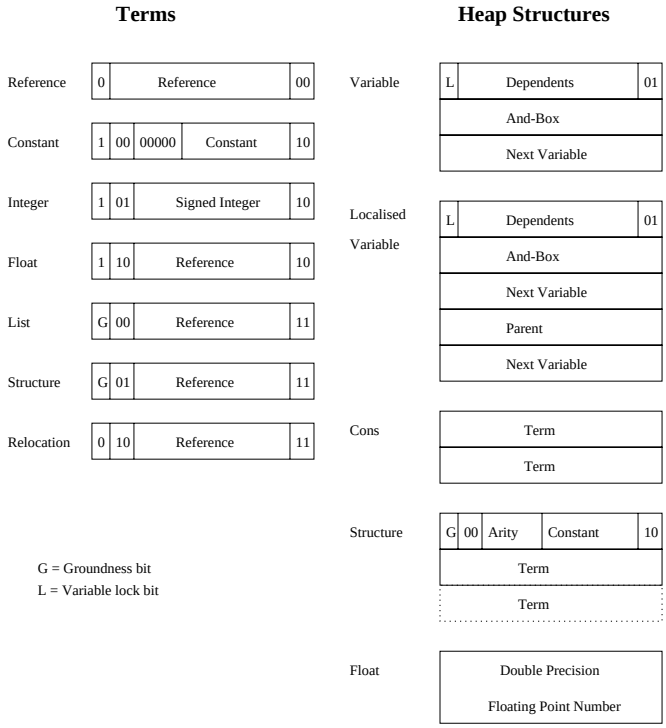

Figure 5.11: DAM Term Representation

5.4.3 Terms

Most terms in the DAM are represented in a similar manner to the WAM. Each term is represented as a 32 bit, tagged word which either contains a reference to another term, an actual value or a reference to a structure on the heap. A summary of the various term structures is shown in figure 5.11.

The lowest two bits of each term supplies a tag value. This tag value allows the references to be represented by the reference address, as references are word-aligned and the low tag bits of a reference are two 0s.

The highest bit indicates whether the term is ground or not. Initially the groundness bit of a term is set to 0, unless it is known from compilation that the term represented is indeed a ground term. If the term is later tested for groundness and found to be ground, then the groundness bit can be set, eliminating the need to re-test for groundness. This approach is similar to that used by DeGroot for restricted and-parallelism [DeG84].

Variables have a lower tag value of 1. It is no longer possible to represent a variable as a reference to itself, as the variable may have a number of dependent localised variables and delayed boxes attached to it. Rather than use an extra word of memory, the reference part of the variable is set to point to the first piece of the dependency list.

Instruction	Description
PutValueXY n,m	Copy the contents of *Xn* to *Ym*.
PutValueYX n,m	Copy the contents of *Yn* to *Xm*.
PutValueXX n,m	Copy the contents of *Xn* to *Xm*.
PutConstant n,$'c'	Put the constant *c* into *Xn*.
PutList n	Construct a cons structure on the heap, and put a reference to this structure into *Xn*.
PutStructure n,$'f'/a	Construct a structure with functor *f* and arity *a* on the heap, and put a reference to this structure into *Xn*.
PutVariableX n	Construct a new variable, local to the current box on the heap and put a reference to the variable into *Xn*. Add this variable to the list of local variables in the box.
PutLiteral n,literal	Put a reference to the ground structure given by *Literal* into *Xn*.
PutListArgument n,a,m	Copy the contents of *Xn* into the *a*th argument (ie. 1 is the head of the list, 2 is the tail) of the list referenced by *Xm*.
PutStructureArgument n,a,m	Copy the contents of *Xn* into the *a*th argument of the structure referenced by *Xm*.

Table 5.5: Put Instructions for the DAM

A non-localised variable is represented on the heap by a structure of three words. While the variable is unbound, the first word contains a variable tag and a reference to the dependency list. When a variable becomes bound, this word is set to the term that the variable is bound to. The second word contains a reference to the and-box that contains the variable in its environment. The following word contains a reference to the next variable in the list of variables local to the and-box.

A localised variable is represented on the heap by a structure containing five words. The first three words are identical to those of a parent variable. Word 4 contains a reference to the immediate parent of the localised variable. The fifth word contains a reference to the next word in the parent dependency list.

Constants in the form of symbolic constants, integers and floats have a lower tag value of 2. The high bit of the term is the groundness bit, which can always be set to 1. The next two bits identify the type of constant: 0 for a symbolic constant, 1 for an integer and 2 for a floating point number. Symbolic constants are represented by an arity (which must be 0) and an index into the symbol table. Integers are represented by signed 27 bit integers. Floating point numbers are represented by a reference to a two-word entry on the heap, containing a double-precision floating point number.

References to lists and other structures have a lower tag value of 3. The groundness bit is set to reflect the known groundness state of the structure referenced. The next two high bits contain the type of reference, either to a list (cons cell) or a structure.

Cons cells are represented on the heap by a pair of terms; the head and tail of the list. Structures are constructed on the heap as a header word, containing the structure name and arity, and a number of following words containing terms.

An additional term is the relocation term, identified by a lower tag of 3 and an upper tag of 2. Relocation terms are used to mark already copied terms on the heap when copying for nondeterminate promotion (see section 5.4.6).

Term Instructions

Term instructions are mostly concerned with performing the unification operations of the constraint system and moving terms between temporary and permanent registers.

Terms are constructed by a sequence of put instructions, similar to the WAM. The DAM avoids read/write unify instructions for reasons given below. Complex structures are built by a sequence of put and put-argument instructions, where sub terms are built and then placed in the appropriate term position. The put

DAM Code	WAM Code
`PutList 0`	
`PutStructure 1,$'f'/2`	`put_structure $'f'/2,X1`
`PutVariable 2`	`unify_variable X2`
`PutStructureArgument 2,1,1`	`unify_value X2`
`PutStructureArgument 2,2,1`	`put_list X0`
`PutListArgument 1,1,0`	`unify_value X1`
`PutConstant 1,$'[]'`	`unify_nil`
`PutListArgument 1,2,0`	

Figure 5.12: Sample DAM Code for Constructing `[f(X, X)]`

instructions supported by the DAM are summarised in table 5.5.

Sample code for constructing the term `[f(X, X)]` is shown in figure 5.12, along with equivalent WAM code for comparison.

Unification is somewhat more complex in the DAM than in the WAM, as variable locking and localisation needs to be taken into consideration. The approach taken by the DAM is to divide the operations of the get-type instructions into two groups: testing and binding instructions. Testing instructions simply test a term to see if it is bound to a particular term, failing the and-box if the test fails or continuing if the box succeeds. If the term being tested is a variable, then the term that the variable will bind to is to be constructed and the variable is bound to the term. Before construction and binding the variable needs to be locked, to ensure no other worker can bind the variable, and localised.

Unification is decomposed in this manner for two reasons: Firstly, in a parallel system, if there are two workers examining a variable then one may be a reader and one a writer. If the reader worker outruns the writer worker while performing a sequence of unify instructions, the reader worker may read garbage. By separating test and bind operations, the DAM can lock a variable, completely construct the term the variable will be bound to and then bind and unlock it in one operation. Secondly, a suitably optimising compiler should be able to detect when locking, localisation and binding operations are not needed. Making these operations separate instructions allows them to be compiled out of the code.

The instructions for unification are summarised in table 5.6. Sample code for the unification `Y = [f(X, X)]` is shown for both the DAM and WAM in figure 5.13. As can be seen, the WAM code is considerably more compact. DAM code for all but the most simple unifications tends to suffer from "code sprawl," as each individual combination of input modes is individually handled. However the DAM code constructs terms efficiently, using put instructions, and avoids the creation of temporary variables, an expensive operation in the DAM.

The DAM also supports some simple arithmetic and term operations as primitives, summarised in table 5.7. These primitives can be combined with various mode-testing instructions (see section 5.4.5), get and put instructions to provide implementations of predicates such as `plus/3` or `functor/3`.

5.4.4 Boxes

Box structures in the DAM are built on the heap and have the formats given in figure 5.14. Each box has a header word that identifies the box as a box using the same tag bits as constants and structure headers. These tags can be used while copying to avoid relocation clashes. The header word also contains the type of box and a locking bit to allow the box to be locked while changing it.

All DAM boxes have the same basic structure: a header word, a set of flag and state bits, pointers to other boxes in the structure and the current program and continuation pointers. The box also contains a reference to the environment for the box and to a vector of arguments. Each box also keeps a count of the number of active workers in the box, the number of children and suspended children. If a box is suspended until a variable becomes bound, the box is linked into the dependents list of that variable. The box also carries a reference to the variable that the box is suspended on; only one variable can be used for suspension at present, more complex suspensions can be built by using predicates with commit guards.

Instruction	Description
GetConstant n,$'c',&label	Test the contents of *Xn*. If a variable, then continue. If the constant given by *c* jump to *label*. Otherwise fail.
GetList n,&label	Test the contents of *Xn*. If a variable, then continue. If a list reference jump to *label*. Otherwise fail.
GetStructure n,$'f'/a,&label	Test the contents of *Xn*. If a variable, then continue. If a structure reference to a structure with functor *f* and arity *a*, jump to *label*. Otherwise fail.
GetVariable n,m	Unify the terms in *Xn* and *Xm*, locking and localising variables as appropriate. This instruction implements a general unification algorithm.
GetListArgument n,a,m	Get the contents of the *a*th argument of the list referenced by *Xn* and put it into *Xm*.
GetStructureArgument n,a,m	Get the contents of the *a*th argument of the structure referenced by *Xn* and put it into *Xm*.
Lock n,&label	Lock the variable referenced by *Xn*. If the variable has become bound, then jump to *label*, otherwise continue.
Localise n,&label	Localise the variable referenced by *Xn*. If *Xn* is localised to a variable, then continue, otherwise jump to *label*.
BindConstant n,$'c'	Bind the variable referenced by *Xn* to the constant *c*.
BindValue n,m	Bind the variable referenced by *Xm* to the term in *Xn*.

Table 5.6: Get Instructions for the DAM

Instruction	Description
Plus l,n,m	*Xl* and *Xn* contain numbers or ground terms that are arithmetic expressions. $Xm := Xl + Xn$.
Times l,n,m	Similar to Plus, except that $Xm := Xl \times Xn$.
Subtract l,n,m	Similar to Plus, except that $Xm := Xl - Xn$.
Divide l,n,m	Similar to Plus, except that $Xm = Xl \mathrm{div} Xn$.
Eval n,m	Evaluate the ground arithmetic expression in *Xn* and place the result in *Xm*.
Equal n,m,&label	*Xn* and *Xm* contain numbers. If $Xn = Xm$ then jump to *label*, otherwise continue. Floats are tested to an accuracy of 10^{-6}.
Less n,m,&label	Similar to Equal, but jump if $Xn < Xm$.
LessEq n,m,&label	Similar to Equal, but jump if $Xn \leq Xm$.
GetArg l,n,m	*Xn* contains a number, *Xl* a structure reference. Copy the contents of the *Xn*th argument of the structure referenced by *Xl* to *Xm*.
PutArg l,n,m	*Xn* contains a number, *Xm* a structure reference. Copy the contents of *Xl* into the *Xn*th argument of the structure referenced by *Xm*.
GetFunctor l,n,m	Take the structure referenced by *Xl* and put the constant functor name into *Xn* and the integer arity into *Xm*.
PutFunctor l,n,m	*Xl* contains a constant and *Xn* contains a number. Construct a structure on the heap with the functor *Xl* and arity *Xn*. Put a reference to the structure in register *Xm*.

Table 5.7: Arithmetic and Term Construction Instructions for the DAM

DAM Code	WAM Code
L1: GetList 0,&L2	
Lock 0,&L1	
Localise 0,&L1	
PutList 1	
PutStructure 2,$'f'/2	
PutVariable 3	
PutStructureArgument 3,1,2	
PutStructureArgument 3,2,2	
PutStructureArgument 2,1,1	
PutConstant 2,$'[]'	
PutStructureArgument 2,2,1	
BindValue 1,0	
Jump &L7	
L2: GetListArgument 0,1,1	get_list X0
L3: GetStructure 1,$'f'/2,&L4	unify_variable X1
Lock 1,&L3	unify_variable X2
Localise 1,&L3	get_structure $'f'/2,X1
PutStructure 2,$'f'/2	unify_variable X3
PutVariable 3	unify_value X3
PutStructureArgument 3,1,2	get_nil X2
PutStructureArgument 3,2,2	
BindValue 2,1	
Jump &L5	
L4: GetStructureArgument 1,1,3	
GetStructureArgument 1,2,2	
GetVariable 2,3	
L5: GetStructureArgument 0,2,1	
L6: GetConstant 1,$'[]',&L7	
Lock 1,&L6	
Localise 1,&L6	
BindConstant 1,$'[]'	
L7:	

Figure 5.13: Sample DAM Code for Unifying Terms

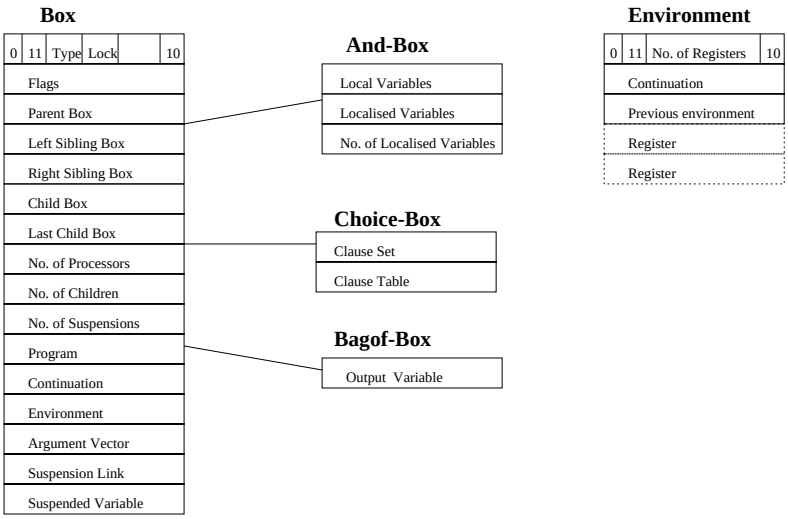

Figure 5.14: DAM Box Representation

And-boxes contain the heads of two lists of variables: those variables local to the and-box and those localised to the and-box. The and-box also maintains a count of the number of localised variables that have not been constrained by a parent variable. Choice-boxes contain the current clause set word and clause table for indexing purposes (see section 5.4.5). Bagof-boxes contain a reference to the list to which completed bagof-calls are added.

The state of the box is represented by a set of flag bits, summarised in table 5.8. The Active, WaitChildren, WaitQuiet, WaitGuard, WaitVar and New bits are mutually exclusive. When a box is killed by some external event, the box may be marked dead or failed, but still be active or queued. This state is used to ensure that the box is not deallocated until all processing has completed.

The WaitVar state indicates that the box has been suspended on a variable. The suspension can come in two forms: a strong suspension indicates that the box is committed to waiting for the variable to become bound, a weak suspension indicates that the box can be woken by the The StrongWait bit indicates a box not available for nondeterminate promotion.

The messages described in section 5.3.5 are implemented by directly modifying the box structure, with locking to ensure mutual exclusion. Messages sent to boxes that have been flagged as dead are abandoned.

An argument vector is used to load the **X** registers with values when a box is taken from the work queue. The argument vector is simply a structure built on the heap, with a suitable arity and functor of `$arg`. Bagof-boxes have an argument vector equal in size to the arguments of the second argument in the bagof call. If the call has no arguments, or the arguments are all stored in an environment, then the pointer to the argument vector is set to a null value.

An environment contains a vector of permanent registers, along with the size of the vector. A reference to the immediate parent environment is also maintained, to allow a return to the parent at the end of the clause. The environment also contains the continuation point for the call. The continuation point is initially set to a null value, and is initialised with the continuation point taken from the parent choice-box upon promotion.

Box Instructions

Box instructions are largely concerned with the creation and removal of boxes and environments. Box instructions are also associated with the flow of control within the computation. The box instructions are summarised in tables 5.9 and 5.10.

A predicate call within the DAM consists of loading the lowest X registers with the arguments to the

Flag	Description
Active	The box is currently active on a processor.
WaitChildren	The box is waiting for all children to complete.
WaitQuiet	The box is waiting until it becomes quiet.
WaitCommit	The box has completed its guard and is waiting to be promoted.
WaitVar	The box is waiting on a variable.
New	The box is a new box, being constructed.
Fail	The box has failed.
Killed	The box has failed or been killed.
Commit	The box has committed to a single clause.
StrongWait	The box is waiting on a variable, with no nondeterminate promotion possible.
Force	The box should ignore any weak waits on variables.
Copy	The box is to be copied
NonDet	The box is for nondeterminate promotion.

Table 5.8: Box Flags for the DAM

DAM Code

```
PutConstant 0,$'a'
PutVariable 1
PutValueYX 1,1
Call 1,'p'/2
```

WAM Code

```
put_constant $'a',X0
put_variable Y1,X1
Call 'p'/2,2
```

Figure 5.15: DAM Code for Calling `p(a, X)`

call and transferring control to the predicate code. The predicate code makes a choice-box if there is local nondeterminism, or if the predicate is going to wait for a variable to be bound. If the predicate call is deterministic, no choice or and-box needs to be constructed. If a predicate is to be spawned to execute on a different worker, the choice-box can be created immediately by the calling box, and queued.

A choice-box creates child and-boxes by executing a series of try instructions. Again an and-box is only created by the and-box code if this proves to be necessary. Alternatively an and-box can be created by a choice-box wishing to queue a choice for parallel execution.

Call and Try instructions may attempt to perform parallel calls or tries, producing and- and or-parallelism. Even if a parallel call or try is requested by the instruction, a parallel call is only executed if there is a idle worker on the work queue waiting for something to be queued. Otherwise the call or try acts as a sequential instruction. A sequential call or try is more efficient than a parallel call, as the box need not necessarily be created, and the box does not have to be queued and the arguments reloaded. Rather than building the parallel call directly, the DAM design assumes that most calls will be sequential calls, and builds the choice-box and argument vector as a special case.

The sequences of abstract instructions for the DAM and WAM for making the call `p(a, X)` is shown in figure 5.15. These sequences are almost identical, the major differences being that the DAM will make the call a parallel call if possible and the WAM reduces the size of the environment where possible.

The code for executing a sequence of choices in both the DAM and WAM is shown in figure 5.16. The major differences between the two sequences is that the DAM separates the operations more directly. The DAM uses a choice-box to perform the same function as a WAM choice-point. However a choice-box can be created remotely, so the creation operation is separated from the try and retry operations.

Instruction	Description
AndBox	If the current box is not an and-box, make a new and-box on the heap and make it the current box.
Allocate n	Make a new environment with *n* permanent variables on the heap, save **CP**, and make it the current environment.
Deallocate	Recover the old **CP** and **E** registers and deallocate the old environment.
Call p,'pr'/a	If *p* is 1, attempt to make a parallel call to `pr/a` by constructing and queueing a choice-box. If *p* is 0, call the predicate directly by setting **P** to the start of the predicate code and **CP** to the next instruction.
Execute 'pr'/a	Set **P** to the start of the predicate `pr/a`, allowing a sequential call at the end of the clause.
Proceed	Return to the calling predicate at the instruction given by **CP**.
Fail	Fail the current box.
WaitChildren	If the current box has child boxes, wait for all child-boxes to complete.
WaitQuiet	If the current box is not quiet, wait for the box to become quiet.
WaitCommit	If the Commit flag is not set for the current box, wait for the parent choice-box to commit to this box.
RequestConditional	Request the parent choice-box to the current box to prune any following sibling boxes.
RequestCommit	Request the parent choice-box to prune any sibling boxes and commit to this box.
Promote	Promote the local variables and localised variables contained in the current and-box to the grandparent and-box.
Raise	Exit the current and-box and make the grandparent and-box the current box.
Collect n,m	Bind the output variable of a bagof-box to *Xm* and make *Xm* the new output variable.
Halt	Similar to a Proceed instruction but intended for the top-level and-boxes in the computation.

Table 5.9: And-Box Instructions for the DAM

DAM Code	WAM Code
`ChoiceBox 2`	
`Try 1,1,&L1`	`try 2,L1`
`Try 2,1,&L2`	`retry L2`
`Try 3,0,&L3`	`trust L3`
`Defer`	

Figure 5.16: DAM Code for Trying a Sequence of Choices

Instruction	Description
ChoiceBox n	If the current box is not a choice-box, make a new argument structure and copy the first *n* temporary registers into the argument structure. Make a new choice-box structure on the heap and make it the current box.
BagofBox	Make a new bagof-box on the heap and make it the current box.
Try c,p,&label	Try a clause starting at *label* if *c* is 0, or *c* is contained in the current clause set. If *p* is 1, try to make a parallel call by constructing and queueing an and-box. If this is a non-parallel call, set the *CP* register to the following instruction and jump to *label*.
TryOne	If the current box is a choice-box, remove the choice box and make the current box the parent and-box.
Defer	If only one child and-box remains for this choice-box, then commit to that and-box and make the and-box active. Otherwise, take the continuation point from the current choice-box and jump to that continuation point.
Quit	Terminate the program.
Slave	Enter the work queue and wait for a piece of queued work.

Table 5.10: Choice-Box Instructions for the DAM

5.4.5 Indexing and Modes

The implementation of NUA-Prolog [PN91] constructed clause sets on the heap. The clause sets could be of any size, and could therefore be adapted to any predicate, no matter how large. This generality carried a considerable cost, both in memory usage and the time taken to search for a singleton bit. A simpler solution is to restrict the clause set to 32 bits and have the clause set contained in the register [Han92, CRR92].

Since the **CS** register is limited to 32 bits it can only handle up to 32 clauses. Most predicates are less than 32 clauses in size so this is rarely a problem. In the case of predicates with more than 32 clauses, a decision tree can be used to perform an initial split on some suitably chosen argument, until the set of possible clauses reduces to 32 or below. If the set of possible clauses stubbornly remains above 32, the initial clauses can be tried until only 32 clauses remain, in which case clause set indexing can be used.

Associated with the clause set is a clause table, pointed to by the **CT** register. The clause table simply consists of a vector of program addresses, corresponding to each clause in the clause set. If the clause set reduces to a single element at any time, the clause table is consulted to see which clause to commit to.

Strong and weak delays are achieved by suspending on a variable. The box is added to the list of dependents and the box then waits. As well as being added to the list of dependents, the box also records the variable that the box has suspended upon. In the case of a box being forced, the box has to be removed from the dependents list of the variable. If the box is copied, the copied box must be added to the dependents list of the copied variable.

Weak waits are forced by setting the Force flag in the box and queueing the box. The box is removed from the variable's dependents list. A box with the Force flag set ignores any weak suspension instructions; strong suspensions are still obeyed, however.

Indexing Instructions

The DAM provides a rich set of instructions for indexing and testing modes, summarised in table 5.11. Sample DAM and WAM code for indexing on a simple program is shown in figure 5.17. The DAM encourages use of specialised forms of clauses for single clause commitment. If the clauses are wait guarded (or conditional or commit guarded and suitably quiet) it is possible to ignore the process of creating an

```
p(a, b) :- true ?  true.
p(b, b) :- true ?  true.
p(b, c) :- true ?  true.
```

DAM Code

```
      ClauseTable 7,(&O1,&O2,&O3)
      SwitchOnTerm 0,&L3,&L1,Fail,Fail
L1:   SwitchOnValue 2,($a:&O1, $b:&L2),Fail
L2:   ClauseSet 6
L3:   SwitchOnTerm 1,&L6,&L4,Fail,Fail
L4:   SwitchOnValue 2,($b:&L5, $c:&O3),Fail
L5:   ClauseSet 3
L6:   ChoiceBox 2
      Try 1,0,&C1,&T2
      GetArguments 2
T2:   Try 2,0,&C2,&T3
      GetArguments 2
T3:   TryLast 3,&C3,&O3
      Defer

O1:   TryOne
      ...
C1:   ...

O2:   TryOne
      ...
C2:   ...

O3:   TryOne
      ...
C3:   ...
```

WAM Code

```
      switch_on_term 0,L1,L2,L4,L4
L1:   try 2,C1
      retry C2
      trust C3
L2:   switch_on_const 0,(a:C1,b:L3)
L3:   try 2,C2
      trust C3

L4:   fail
C1:   ...

C2:   ...

C3:   ...
```

Figure 5.17: DAM Code for Indexing

Instruction	Description
SwitchOnTerm n,&v,&c,&l,&s	Examine the contents of *Xn* and jump to *v* if a variable, *c* if a constant, integer or floating point value, *l* if a list reference and *s* if a structure reference.
SwitchOnValue n,&table,&fail	Examine the contents of *Xn* which is a constant or structure reference. Use the value of the constant or functor and arity of the structure as a key to the hash table, *table*. If the value is found in the hash table, jump to the corresponding program address, otherwise jump to *fail*.
ClauseTable set,&table	Make the current clause set *set* (an integer) and the current clause table *table*.
ClauseSet set	Intersect the current clause set with *set* (an integer). If the resulting clause set is empty, fail the current box. If the resulting clause set has a single element, jump to the corresponding entry in the clause table.
JumpVar n,&label	If *Xn* contains a reference to a variable, jump to *label*.
JumpGround n,m,&label	If *Xn* contains a ground structure, then jump to *label*. Otherwise, place a reference the first unbound variable in the structure into *Xm*.
Suspend n,w,&label	Strongly suspend on the variable referenced by *Xn*, restarting at *label* when the variable becomes bound. If w is 1, then weakly suspend or skip this instruction if the current box has the Force flag bit set.

Table 5.11: Indexing and Mode Instructions for the DAM

and-box, localising variables and promoting, since all of this is going to happen unconditionally.

5.4.6 Nondeterminate Promotion and Copying

Nondeterminate promotion is handled in the DAM by queueing the choice-box that is to be promoted with the NonDet flag bit set. The essential approach of the AKL is to be "or-phobic," wherever possible, so nondeterminate promotions are never executed immediately.

When a worker takes a box with the NonDet flag from the queue, it performs a nondeterminate promotion on the box rather than executing the box immediately. Eventually the worker will execute the child and-box that has been chosen for promotion.

Stream and-parallelism tends to create scattered blocks of related data, as the various and-computations interleave. The binding array and copying models of or-parallelism (see sections 2.1.2 and 2.1.4) rely on bindings being available as a contiguous block for efficiency while copying.

The multi-sequential machine model (see section 2.1.3) relies on workers shadowing the initial branches of an or-parallel computation. These workers would be more profitably employed performing any and-parallelism that was available. The Delphi model requires special pre-processing and worker re-synchronisation appears to be potentially expensive.

The hash window model (see section 2.1.1) has been used successfully in the ParAKL machine [MD93]. However, use of localised variables in the DAM means that there are no readily available hash windows for maintaining multiple bindings.

The DAM copies box-trees in a manner similar to copying garbage collection. This approach is also used in the AGENTS abstract machine [Jan94] and the Penny system [MA96]. Each object that is part of the structure being copied is allocated a new block of memory and copied across. As each structure is copied its first word is replaced by a relocation term, giving the address of the copied object. Further attempts to copy an already relocated object simply use this relocation term. Each relocation term is trailed and the trail unwound at the end of copying.

nrev(1000)	Naive reverse with 1000 elements
qsort(2500)	Quicksort with 2500 elements
fib(25)	Fibonacci number of 25
tree(17)	Generate a balanced binary tree of depth 17
subset(15)	Generate all subsets of a set of 15 elements
encap(7)	Generate a cross product of two subsets of 7 elements.
filter(1000)	Filter a list of 1000 elements by selecting the elements contained in another list.
and(50000)	Test the and boolean operator with various combinations of modes.

Table 5.12: Benchmarks

Starting at the head of the sub-tree, a duplicate of each box is constructed, copying the flags, program and continuation pointers. And-boxes make copies of their environments, and local and localised variable lists. Or-boxes make copies of their arguments.

Local variable lists need only have those local variables that are still unbound copied; bound variables have their bindings copied when the and-boxes environment is copied. Localised variable lists are copied in two passes, ignoring any localised variables that have been entailed by their parents. Initially copies of each localised variable are made and the original variables given relocation pointers. After all localised variables have been copied, each new localised variable has the term that it is bound to copied.

Terms are copied recursively, with each term being checked for groundness before a copy is made. Ground terms do not need to be copied. The use of groundness bits means that a ground term is only ever checked once for groundness, making the copying of ground terms a great deal more efficient.

Child boxes are only copied after the environment and variables of the parent box has been copied, meaning that all references to variables outside the scope of child boxes have been relocated before-hand.

Child boxes can be copied in parallel since no child box can have bindings referring to a sibling box. Boxes for parallel copying are queued with the Copy flag-bit set, and the copy is made on the heap of the worker which acquires the box from the queue. The copying worker can also trail relocation references on its own trial, and unwind the trail at the end of the copy, since the copying of a child box cannot have an effect on the environment of the parent box. To avoid queueing marginal prospects for copying, only those child-boxes with at least one child-box themselves are considered for parallel copying.

5.5 Performance

This section presents some performance results for the DAM. The benchmarks are summarised in table 5.12 and code for the benchmarks can be found in appendix A. **nrev(1000)** and **qsort(2500)** test dependent and-parallelism. **fib(25)** and **tree(17)** test determinate independent and-parallelism. **subset(15)** tests or-parallelism. **encap(7)** tests nondeterminate independent and-parallelism. **filter(1000)** tests deep guard performance. **and(50000)** tests clause set indexing.

Running more standard nondeterminate benchmarks on the DAM, such as the SEND + MORE = MONEY problem reveals a peculiarity with the scheduling of nondeterminism on the DAM. The work queue on the DAM tends to produce behaviour similar to breadth-first search, as both a nondeterminate promotion and the remainder of the promoted choice-box are scheduled for execution. As a result, the DAM takes almost as long to compute 1 solution to the SEND + MORE = MONEY problem as it takes to compute all solutions.

For purposes of comparison, the benchmarks were run on the following implementations:

NU-Prolog [ZT86] An example of a sequential Prolog system built around the WAM. Version 1.6.5 was used.

AGENTS [Jan94] The SICS sequential AGENTS abstract machine, version 1.0.

Benchmark	NU-Prolog	AGENTS	Penny	DAM	DAM (no locks)
nrev(1000)	2100	6800	8300	8400	6500
qsort(2500)	1300	3000	5200	4800	3400
fib(25)	4300	4800	5800	4900	4200
tree(17)	2900	3900	5000	5300	4400
filter(1000)	830	2300	5000	18000	17000
subset(15)	430	13000	7600	13000	12000
encap(7)	300	5000	2500	3100	3000
and(50000)	2600	6000	8500	7800	5900
All results in ms, $\pm 10\%$					

Table 5.13: Single Processor Performance of the DAM

Penny [MA96] The experimental SICS parallel AKL abstract machine.

DAM Doug's Abstract Machine.

All systems were compiled using gcc or g++ version 2.7.2.1 using the -O2 optimisation level for gcc and the -O3 optimisation level for g++.[2] The benchmarks were run on a Sun SparcServer 1000 with four 50MHz processors on a 40MHz bus running the Solaris 2.3 (SunOS 5.3) operating system. All benchmarks were run using the RT (real time) scheduling class, to ensure the highest CPU availability possible.

Both parallel systems tended to produce variable times for most benchmarks, with the variability becoming more pronounced as the number of processors increased. The times quoted are the mean of three consistent trials, with there being approximately 10% variation between most trials.

Single processor performance for the DAM for a set of benchmarks is shown in table 5.13. In addition to the normal DAM, a sequential version of the DAM where all locking was removed was also tested.

As might be expected, the sequential NU-Prolog system outperforms all the AKL systems. In most cases, the performance of NU-Prolog is 3-4 times that of the best AKL system. The additional overhead caused by the manipulation of boxes and handling of local/non-local variables would reasonably cost that much.

When considerable non-determinism is present, in the **subset(15)** and **encap(7)** benchmarks, NU-Prolog outperforms the AKL systems by anywhere from 7 to 40 times. Since NU-Prolog uses backtracking, as opposed to the copying mechanisms used the AKL systems, the massive improvement in performance in a nondeterminate program can be attributed to the superiority of backtracking to copying; when copying, the DAM needs to trail anything that it is relocating as well as copying it — clearly trailing, copying and unwinding must take longer than simply trailing and unwinding.

The **fib(25)** benchmark results are the results where the AKL most closely match the performance of NU-Prolog. This benchmark involves a large number of arithmetic operations; operations that are not usually part of the core abstract machine. In this case, all implementations are more likely to be equal. Since the **tree(17)** benchmark is very similar to **fib(25)**, but involves less arithmetic and shows a similar performance difference between the Prolog and AKL implementations, the arithmetic operations are likely to be the prime cause of this anomaly.

In general, the DAM tends to outperform Penny on the simpler benchmarks, where box construction can be avoided. On the simpler benchmarks, the DAM with no locks is roughly equal to AGENTS in performance. In the **filter(1000)** benchmark, where deep guards and a great deal of box creation occurs, the DAM is much more inefficient than the AGENTS and Penny systems.

The **and(50000)** benchmark shows similar performance to the AGENTS system when the overhead of locks is removed. The **and(50000)** also shows the expense of box creation; profiling of the program showed that 12% of time was spent creating and removing choice-boxes.

In the nondeterminate benchmarks, **subset(15)** and **fib(7)**, the DAM and AGENTS implementations tend to show similar performance; an expected result, since nondeterminate promotion tends to use a lot of

[2] Optimisation levels were left as set by the authors of the programs. -O3 level inclines functions declared as such, a practise usually ignored in C, where `#define`s are used instead, but encouraged in C++

		Processors			
Benchmark	System	1	2	3	4
nrev(1000)	DAM	8400	5600	3800	3600
	Penny	8300	5300	4400	3700
qsort(2500)	DAM	4800	3800	3600	3700
	Penny	5200	2900	1600	1500
fib(25)	DAM	4900	3200	2500	2400
	Penny	5900	3600	2500	1700
tree(17)	DAM	5300	4200	3100	3600
	Penny	5000	2600	1900	1600
subset(15)	DAM	13000	11000	9200	9200
	Penny	7600	5500	4900	4600
encap(7)	DAM	3100	2400	2100	2300
	Penny	2500	1900	1700	1700
filter(1000)	DAM	19000	18000	19000	20000
	Penny	5000	5300	5200	5200
and(50000)	DAM	7800	14000	24000	26000
	Penny	8500	16000	19000	20000
All results in ms, $\pm 10\%$					

Table 5.14: Parallel Performance of the DAM

time copying. Penny outperforms the DAM and AGENTS on non-determinate benchmarks. Since Penny uses a similar copying algorithm to the other AKL systems, this difference is more likely to be a result of a more highly optimised implementation than some underlying architectural difference.

For programs that contain little nondeterminism, there is approximately 20% performance cost, largely traceable to the costs of locking. Nondeterminate promotion is designed to avoid most locking and so does not suffer so much from the performance penalty. With locking removed, the DAM performs comparably to the AGENTS machine and Penny; the extra speed of the DAM in some cases can be explained by the inlined arithmetic functions available on the DAM.

Parallel processor performance for the DAM is summarised in table 5.14. Performance results for 4 processors should be regarded with some caution, as the system used for the benchmarks has 4 processors, and contention with the operating system is possible when all processors are being used.

The DAM and Penny tend to show roughly the same 1 processor performance for the simpler determinate programs, but a more linear speed-up as more processors are added. In most cases, the parallel speed-up for the is nowhere near linear, a disappointing result. There seem to be two reasons for this result:

Profiling the DAM shows that the ChoiceBox instruction is very expensive, taking over 100 times the amount of time that the simpler instructions take. The dependent and-parallel programs tend to use choiceboxes to synchronise and so pay the penalty.

The work queue can become saturated with tiny pieces of work, leading to a granularity problem. This is the case with the fib(25) test, since many tiny Fibonacci numbers are computed towards the end of the computation, and many of these will be queued in parallel. The **and(50000)** benchmark contains a great deal of fragmentary parallelism. Most calls to `and/3` are simple and incur little overhead if performed sequentially. The subset(15) and encap(7) programs tend to perform many small nondeterminate promotions, leading to contention at the work queue.

The **filter(1000)** benchmark has little inherent parallelism, and so little or no parallel speed-up should be expected. In fact, the parallel overhead tends to cause both systems to perform worse on essentially sequential programs when more processors are added.

No memory benchmarks have been included. Memory benchmarks are difficult to come by, as they are generally not published, and measurement in systems that do not supply statistics is very difficult. NU-Prolog can be expected to be generally most memory conservative, since it does not need box and suspension structures for the benchmarks described here. The DAM can be expected to use the most memory.

The need for large variables (over one cell in size) and the size of box structures tends to make the DAM memory-wasteful. Thy middle ground between NU-Prolog and the DAM can be expected to be occupied by AGENTS and Penny.

5.6 Related Work

The closest relation to the DAM is the Penny parallel machine and its sequential predecessor, the AGENTs abstract machine [MA96, Jan94]. Both these machines use a system of configuration-stacks to control the computation, rather than the box-based approach of the DAM. In the Penny architecture, workers move from box to box in a localised fashion, taking work from an immediate parent or child box; in contrast, the DAM uses a queue and activates a box by placing it on the queue. Scheduling involves other workers acquiring outstanding tasks from an owning worker, rather than the DAM's approach of workers picking up any outstanding work from a common queue. The tendency of the DAM to perform breadth-first search, and the granularity problems that it experiences suggests that the DAM should use a stacking-based scheduler for at least some of its work.

The ParAKL system [MD93] uses hash tables to maintain multiple binding environments.

The Andorra-I system [SCWY91a] is intended for use with the basic Andorra model and therefore does not need the complications of multiple binding environments. Similarly, Pandora [Bah91] is based on Parlog and the JAM and does not need to consider such complications.

Chapter 6

An AKL Compiler

This chapter covers the compiler for the DAM. The DAM contains several expensive operations: the creation and destruction of boxes, locking and localising variables, and copying parts of the box-tree during non-determinate promotion. A useful compiler for the DAM needs to be able to minimise the number of times these operations are performed.

A compiler is essentially a program that translates a program in one language, the source language, into a program in another language, the target language, with the same semantics. The target language is usually a lower-level language than the source language. An optimising compiler does this translation in such a way as to minimise some characteristic of the output program, such as the time the program will take to run or the memory space that the program takes.

A block diagram for the DAM compiler for the is shown in figure 6.1. This compiler follows the typical structure of a logic programming compiler. Individual clauses are compiled into streams of abstract instructions. All clauses for a predicate are then analysed and a *prelude* constructed: a set of instructions controlling indexing and nondeterminism. The instruction stream is passed through a peephole optimiser and the resulting stream is then written to a file. The abstract interpretation step is not part of a normal compiler, but is part of the DAM compiler. The abstract interpretation step allows the program to be analysed and information about the nature of predicates passed to the clause compilation and indexing parts of the compiler, with the aim of producing a more efficient compilation.

An optimising compiler gathers information about the expected behaviour of a program. This information is then used to tailor the product of the compiler so that it executes in a more efficient manner. Usually compilers are *local;* the compiler works on one unit — a clause or predicate — at a time, and does not use information gathered from the compilation of other units. Alternatively a compiler can be *global* to varying degrees. Global compilers gather information about all parts of a program that are available, and use that information to optimise the compiler output. Possible global optimisations are:

Goal Ordering: Computations are most efficient when the goals in a clause are ordered so that data flows from producers to consumers. Although the AKL execution model is largely insensitive to goal ordering issues, the DAM is most efficient when predicates are called with suitably bound variables; creating a choice-box and delaying a call are expensive operations and should be avoided.

Determinacy: If a call to a predicate is determinate, it may not be necessary to create a choice-box and and-box for a local environment. Determinacy can be achieved in two ways. Firstly if the predicate is known to be dependent on some arguments being bound to be determinate, the call can be delayed until the appropriate arguments are bound. Secondly, complete indexing of clauses can also help detect determinate goals.

This chapter concentrates on the more interesting aspects of the compiler; those parts where the compiler deviates from simply converting goals into lists of unification and call instructions. The compiler is intended to optimise the DAM instructions produced, eliminating the most costly operations where possible. Optimisation is achieved at two levels: an abstract interpretation which gathers the types and modes of each predicate, and in-line optimisations produced by gathering requirements from the future stages of the meta-interpretation.

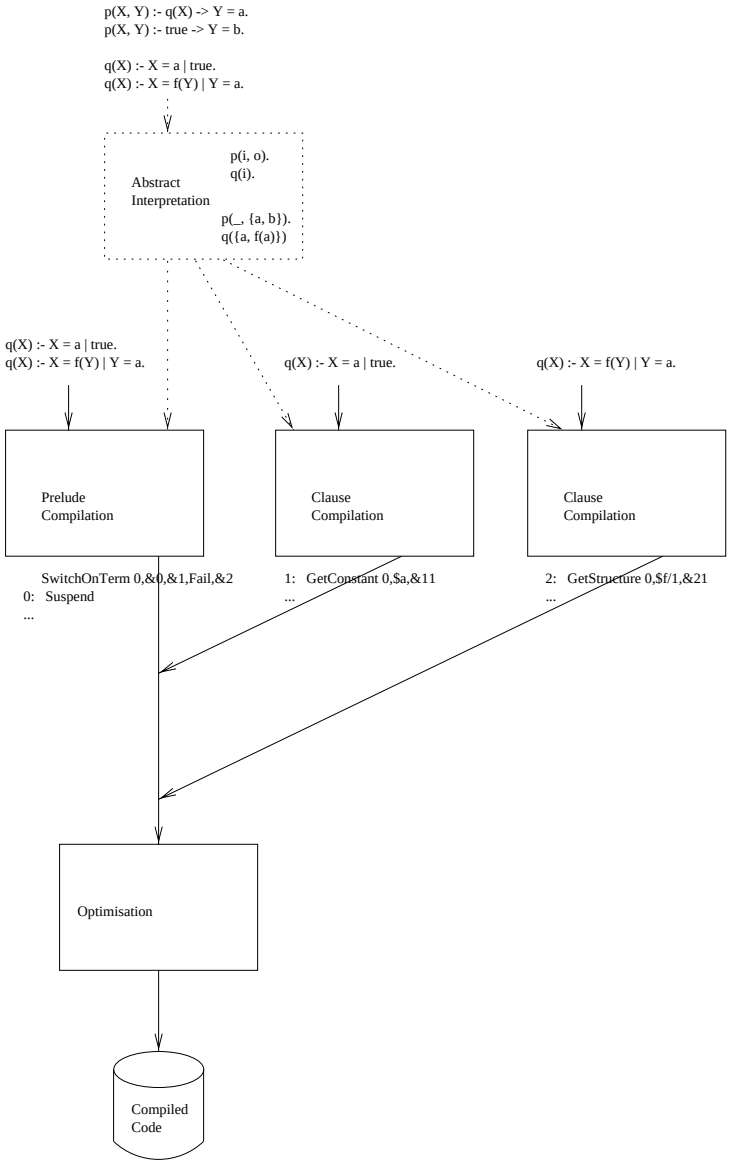

Figure 6.1: Compiler Architecture

6.1 Abstract Interpretation

Abstract interpretation was introduced by Cousot and Cousot [CC77] as a method of formally deriving program properties. Using an abstract interpretation involves an approximate execution of the program, which is guaranteed both to halt and work with a superset of the possible values of the actual program. The information derived from the approximate execution can then be used to annotate the program in several ways, giving a compiler clues as to the most efficient way to compile pieces of program.

The simple semantics of logic programming languages make them ideal for abstract interpretation, and abstract interpretation models have been used for several purposes, such as variable dependence tests for independent and-parallelism [MH92], specialising versions of predicates [Win92] and compile-time garbage collection [MWB90].

An abstract interpretation involves two posets, a *concrete domain*, denoted by $< E, \leq >$, that is an accurate representation of the computation and an *abstract domain* $< D, \sqsubseteq >$ that represents some useful abstraction of the concrete domain. The two domains are related by means of a concretisation function $\gamma : D \rightarrow E$ and an abstraction function $\alpha : E \rightarrow D$. Each operation s in the concrete domain, is mapped onto an abstract operation s' in the abstract domain.

To be a safe abstract interpretation, the domains must be *adjoined* [MS92].

Definition 6.1.1 Let D and E be complete lattices. The monotonic functions $\gamma : D \rightarrow E$ and $\alpha : E \rightarrow D$ are *adjoined* iff $\forall d \in D, d = \alpha(\gamma(d))$ and $\forall e \in E, e \leq \gamma(\alpha(e))$.

In most logic programming languages, abstract interpretations can be built to model either top-down or bottom-up execution (eg. SLD-resolution and the T_P fixpoint semantics respectively in the case of Prolog). An example of top-down abstract interpretations is the general framework described by Bruynooghe et al. [BJ88], Another example is multiple specialisation [JLW90], which allows the creation of multiple, specialised versions of a predicate, selected by run-time tests. Most specific logic programs [MNL88] can be used to generate more efficient versions of programs; an example of bottom up abstract interpretation is the generation of most specific programs [MS92].

Abstract interpretations may also be based on a parallel or sequential model of interpretation. Bottom-up execution is implicitly parallel, as each new fact can be computed independently of all other possible facts. Bottom-up abstract interpretations therefore, are also implicitly parallel. Top-down execution models, especially committed-choice languages, may be sensitive to deadlock and race conditions. As a result, the abstract interpretation must take account of these possibilities. An example of a top-down abstract interpretation that handles parallelism is Codish's suspension analysis [Cod91].

The main existing form of abstract interpretation for the AKL is that of Sjöland and Sahlin [SS95] which uses top down interpretation to derive a set of pre- and post- call domain equations which can be solved. A variety of methods for solving the equations are examined by Schön [Sch95]. An alternative to directly using top-down abstract interpretation for top-down, parallel execution models is to build an equivalent bottom-up execution model and use that model to derive the abstract interpretation. This technique has the advantage that it side-steps the inherent complexities of explicitly modelling parallelism, while producing the same results in some areas. An example of this technique is the abstract interpretation developed by Foster and Winsborough for Strand [FW91], used to avoid copying of modified data structures.

The compiler described here uses abstract interpretation to provide more accurate indexing of clauses throughout the program, and to provide accurate mode information for determining call orders and call delaying. The bottom-up abstract interpretation used in this compiler uses the fixpoint semantics developed in chapter 4, extended to allow explicit mode information as a concrete domain.

6.1.1 Partitioning the Program

The fixpoint semantics in chapter 4 rely on a program being guard stratified. Before being able to perform any abstract interpretation the program must be partitioned into guard-stratified layers. Stratifications can be built by building a call graph between all the predicates in the program. The program is then partitioned in the following manner:

If p and q are two predicates, then $p \leq q$ is defined recursively by: $p \leq q$ if any clause of q contains a direct reference to p, and if $p \leq q$ and $q \leq r$ then $p \leq r$. The partition of a program P consists of the set

of equivalence classes of the predicates in P with respect to $\leq$. A program P is guard stratified if, for all clauses in P, the predicates in the guard of all clauses of q, p have $p < q$.

As an example, the program

```
p1(X)  :-  true  ?  p4(X)
p1(X)  :-  p2(X, Y)  ?  p4(Y)

p2(f(X), Y)  :-  true  -> X = Y.
p2(g(X), Y)  :-  p3(X)  -> X = Y.

p3(a)  :-  true  ?  true.
p3(b)  :-  true  ?  true.

p4(X)  :-  p3(X) ? true.
p4(c)  :-  true ?  p1(a).
```

has the partition $\{\{p1, p4\}, \{p2\}, \{p3\}\}$. The program is clearly guard stratified.

The partition of a program P can be built by constructing a directed graph of all calls in the program and then computing the transitive closure. All arcs on the graph where both (p, q) and (q, p) exist are part of the same equivalence class.

In the above example program, the directed graph consists of the edges

$$\{(p1, p2), (p1, p4), (p2, p3), (p4, p1), (p4, p3)\}$$

with a transitive closure of

$$\{(p1, p2), (p1, p3), (p1, p4), (p2, p3), (p4, p1), (p4, p2), (p4, p3)\}$$

Clearly, the only pair of arcs here is $(p1, p4), (p4, p1)$, making the equivalence classes $\{\{p1, p4\}, \{p2\}, \{p3\}\}$

Checking that the program is guard stratified essentially means ensuring that all predicates in each equivalence class do not call each other in their guards. If a program is not guard-stratified, a warning is issued, and those predicates which are not guard stratified are excluded from the abstract interpretation — they are assumed to have the most general type and mode.

6.1.2 Determining Types

A major aim of applying abstract interpretation to the AKL is to determine the types of terms that variables may be bound to. Type determination leads to more efficient indexing of clauses and earlier detection of determinacy. For example in the program

```
p(X)  :-  q1(X) -> r(X).
p(X)  :-  q2(X) -> s(X).

q1(f(a)).
q1(f(b)).

q2(f(b)).
q2(g(b)).
```

the predicate `p/1` needs its argument to be bound to be determinate. A simple indexing algorithm would not be able to deduce that this is possible. However an indexing algorithm that has the expected success patterns of `q1/1` and `q2/1` will be able to more efficiently index `p/1`, eliminating useless speculative computation.

The concrete domain for types is the domain of constraints over Herbrand equality. In order to aid analysis a normal form is chosen. The approach taken throughout the abstractions developed below is to group terms together by their position in a term tree. *Traces* provide a consistent way of referring to the position of sub-terms within terms.

Definition 6.1.2 A *trace* is a sequence, recursively defined as follows: the empty trace, ϵ, is a trace and $i.t$, where $i \in \mathbf{N}$ and t is a trace, is a trace. $\mathbf{N}$ is the domain of natural numbers.

The domain of traces is denoted by $\mathcal{TR}$.

Traces are ordered by

$$\begin{array}{ll} \epsilon \leq t & \\ s_h.s_t \leq t_h.t_t & \text{if } s_h < t_h \\ s_h.s_t \leq t_h.t_t & \text{if } s_h = t_h \text{ and } s_t \leq t_t \end{array}$$

The concatenation of two traces s and t is denoted by $s \cdot t$. By an abuse of notation the concatenation of s and $i.\epsilon$ is also written as $s \cdot i$.

The application of a trace t to a term r, written as $r \diamond t$ is defined as:

$$r \diamond t = \begin{cases} r & \text{if } t = \epsilon \\ r_i \diamond t_t & \text{if } t = i.t_t, r = f(r_1, \ldots, r_n) \text{ and } 1 \leq i \leq n \\ \bot & \text{otherwise} \end{cases}$$

Definition 6.1.3 A constraint is in *trace conjunctive form* if it consists of a set of constraints: $\theta_1 \wedge \cdots \wedge \theta_n$ where each θ_i is of the form $V_s = c$, $V_s = f(V_{s.1}, \ldots, V_{s.m})$ or $V_s = V_t$ where $t \leq s$.

A constraint *constrains* V_s if there is some θ_i $V_s = c$ or $V_s = f(V_{s.1}, \ldots, V_{s.m})$, or $V_s = V_t$ where the constraint constrains V_t.

The domain of constraints in trace conjunctive form is denoted by $\mathcal{CF}$.

A set of constraints is in *trace disjunctive form* if it is in the form $\sigma_1 \vee \cdots \vee \sigma_l$ where each σ_i is in trace conjunctive form.

Lemma 6.1.1 Any constraint consisting of equalities of terms can be put in trace disjunctive form.

Proof Any constraint can be put into disjunctive normal form. We begin by renaming all variables to be of the form $V_{i.\epsilon}$. Any equalities of the form $f(S_1, \ldots, S_n) = f(T_1, \ldots, T_n)$ can be replaced by $S_1 = T_1 \wedge \cdots \wedge S_n = T_n$. Any equalities of the form $V_t = f(T_1, \ldots, T_n)$ can then be replaced by $V_t = f(V_{t.1}, \ldots, V_{t.n}) \wedge V_{t.1} = T_1 \wedge \cdots \wedge V_{t.n} = T_n$ Any equalities of the form $V_s = V_t$ can be rewritten as $V_t = V_s$ if $s \leq t$. □

Trace form provides a convenient canonical form for the analysis of types. Using trace form we can define a type system that is specialised towards the instantiation patterns of variables.

Definition 6.1.4 A *broad type* is one of the following elements:

1. A lattice of base types such as *atom*, the set of all atoms, *term* the set of all non-atomic terms or *integer* the set of all integers. Two base types that must exist are $\top$, the universal type indicating all terms, and $\bot$, the empty type. Base types have a function ψ defined to be the set of functor/arity pairs that can be represented by that base type (e.g. $\psi(integer) = \{0/0, 1/0, -1/0, 2/0, -2/0, \ldots\}$, where integers, as constants, have an arity of 0)

2. $F(T_1, \ldots, T_n)$, where $F = \{f_1/n_1, \ldots, f_m/n_m\}$ is a set of functor/arity pairs, $n = \max(n_1, \ldots, n_m)$ and each $T_i, 1 \leq i \leq n$ is a broad type.

The symbol $\mathcal{B}$ is used to represent the set of all broad types.

Broad types have a similar trace application operator to terms (see definition 6.1.2).

The functor set for a broad type is defined as

$$\Psi(T) = \begin{cases} F & \text{if } T = F(T_1, \ldots, T_n) \\ \psi(T) & \text{if } T \text{ is a base type} \end{cases}$$

Broad types represent a set of terms, with all possible terms that can be built from the type tree. As an example, the broad type $\{f/1, g/2\}(\{a/0, b/0\}, \{a/0, c/0\})$ represents the set of terms $\{f(a), f(b), g(a, a), g(a, c), g(b, a), g(b, c)\}$.

The broad type $\{[]/0, ./2\}(\top, \{[]/0, ./2\}(\top, \top))$ represents the set of terms $\{[], .(\_, []), .(\_, .(\_, \_))\}$ or the first three possible lists.

Broad types are both less flexible and less precise than the regular types used by (for example) Zobel [Zob90]. It is not possible to represent a general recursive type, such as $list = \{[], .(\top, list)\}$ using broad types. It is also not possible to separate arguments in broad types, e.g. $\{f(\{a, b\}), g(\{c, d\})\}$. However broad types are well suited to the role they play in the AKL; broad types provide a suitable abstraction for the purposes of clause set indexing.

Definition 6.1.5 The *unification* of two broad types T and S, written as $T \wedge S$ is given by

1. If $T = \top$ then S.
2. If T and S are base types glb(S, T).
3. If $T = F(T_1, \ldots, T_n)$ and S is a base type, then $\{f/n : f/n \in F_T \text{and} f/n \in \psi(S)\}$.
4. If $T = F(T_1, \ldots, T_{n_T})$ and $S = G(S_1, \ldots, S_{n_S})$ then $(F \cap G)(T_1 \wedge S_1, \ldots, T_n \wedge S_n)$ where $n = \min(n_T, n_S)$. If any of $T_i \wedge S_i = \perp$ or $F \cap G = \emptyset$ then this is equivalent to $\perp$.

The *union* of two broad types T and S, written as $T \vee S$ is given by

1. If $T = \perp$ then S.
2. If T and S are base types lub(S, T).
3. If $T = F(T_1, \ldots, T_n)$ and S is a base type, then R where R is a base type and $T \leq R$ and $S \leq R$.
4. If $T = F(T_1, \ldots, T_{n_T})$ and $S = G(S_1, \ldots, S_{n_S})$ then $(F \cup G)(T_1 \vee S_1, \ldots, T_n \vee S_n)$ where $n = \max(n_T, n_S)$, and $T_i = \perp$ if $i > n_T$, similarly for S.

Both unification and union are commutative operators.

Examples of unification and union of broad types are:

$$\{f/1, g/2\}(\top, integer) \wedge \{f/1\}(atom) = \{f/1\}(atom)$$

and

$$\{f/1, g/2\}(\top, integer) \vee \{f/1\}(atom) = \{f/1, g/2\}(\top, integer)$$

Definition 6.1.6 The *depth* of a broad type is a function depth : $\mathcal{B} \rightarrow \mathbf{N}$, defined as follows:

$$\text{depth}(T) = \begin{cases} 1 & \text{if } T \text{ is a base type} \\ 1 + \max(1, \text{depth}(T_i)) & \text{if } T = F(T_1, \ldots, T_n). \end{cases}$$

The concrete domain, E, for this type abstraction is the set of constraints in trace disjunctive form. For two elements $e_1, e_2 \in E$, we have an ordering defined as $e_1 \leq e_2$ if $e_1 \rightarrow e_2$. Our abstract domain is over the set of mappings $D = \wp(\mathcal{V} \rightarrow \mathcal{B})$ where a variable is mapped onto a broad type. Only base variables (ie. variables with a single element trace) need to be included in the mapping.

We can define the concretisation functions and abstraction functions for moving between broad types and constraints in trace normal form in the following way:

Definition 6.1.7 The *concretisation function* $\gamma : \mathcal{V} \times \mathcal{B} \rightarrow E$ is defined, for $\gamma(V_t, d)$, as:

1. The constraint $isBase(V_t)$, if $d = B$ where B is some base type. The base types $\top$ and $\perp$ are mapped onto the constraints true and false respectively.

2. The constraint

$$\left(\begin{array}{c}(V_t = f_1(V_{t\cdot 1}, \ldots, V_{t\cdot m_1}) \\ \vee \cdots \vee \\ (V_t = f_n(V_{t\cdot 1}, \ldots, V_{t\cdot m_n})\end{array}\right) \wedge \gamma(V_{t\cdot 1}, d_1) \wedge \cdots \wedge \wedge \gamma(V_{t\cdot m}, d_m)$$

if $d = \{f_1/m_1, \ldots, f_n/m_n\}(d_1, \ldots, d_m)$.

The concretisation function $\gamma : D \rightarrow E$ is defined as $\gamma(d) = \bigwedge_{V \in dom(d)} \gamma(V, d(V))$.

Definition 6.1.8 The abstraction function $\alpha : \mathcal{V} \times E \rightarrow \mathcal{B}$ is defined as

$$\alpha(V_t, e) = \begin{cases} \alpha(V_t, e_1) \vee \cdots \vee \alpha(V_t, e_n) & \text{if } e = e_1 \vee \cdots \vee e_n \\ \alpha(V_s, e) & \text{if } e = e_1 \wedge \cdots \wedge V_t = V_s \wedge \cdots \wedge e_n \\ b & \text{if } e = e_1 \wedge \cdots \wedge isBase(V_t) \wedge \cdots \wedge e_n \\ \{c/0\}() & \text{if } e = e_1 \wedge \cdots \wedge V_t = c \wedge \cdots \wedge e_n \\ \{f/m\}(\alpha(V_{t\cdot 1}, e), \ldots, \alpha(V_{t\cdot m}, e)) & \text{if } e = e_1 \wedge \cdots \wedge \\ & V_t = f(V_{t\cdot 1}, \ldots, V_{t\cdot m}) \wedge \cdots \wedge e_n \end{cases}$$

b is the base type corresponding to $isBase(V)$.

The abstraction function $\alpha : E \rightarrow D$ is defined as $\alpha(e)(V) = \alpha(V, e)$ for all V of the form $V_{i.\epsilon}$.

Lemma 6.1.2 The functions α and γ as defined in definitions 6.1.8 and 6.1.7 are adjoined.

Proof The proof consists of an induction on the depth of the abstract domain element for each variable. Since the definition of α decouples variables, only individual variables need to be considered.

If depth$(T) = 1$ then T is a base type. In such a case $\gamma(V, T) = isBase(V)$ and $\alpha(V, isBase(V)) = T$. Similarly $\gamma(V, \alpha(V, isBase(V))) = isBase(V)$.

Suppose the proposition holds for all types of depth less than n, then if $T = F(T_1, \ldots, T_m)$ where depth$(T_i) < n, 1 \leq i \leq m$.

$\gamma(V_t, T)) = \left(\bigvee_{f/m \in F} V = f(V_{t\cdot 1}, \ldots, V_{t\cdot m})\right) \wedge \gamma(V_{t\cdot 1}, T_1) \wedge \cdots \wedge \gamma(V_{t\cdot m})$ from the definition of γ. This function can be distributed across the disjunctions to give a normal form. From the definition of α we have
$\alpha(V_t, \gamma(V_t, T)) = \bigvee_{f/m \in F} \{f/m\}(\alpha(V_{t\cdot 1}, \gamma(V_{t\cdot 1}, T_1)), \ldots, \alpha(V_{t\cdot m}, \gamma(V_{t\cdot 1}, T_m))$ which since the proposition holds for each $T_1, \ldots, T_m$ and from the definition of union of broad types we have $\alpha(V_t, \gamma(V_t, T)) = F(T_1, \ldots, T_m) = T$.

To prove that $\gamma(\alpha(e)) \leq e$ we use the constraint property that $(c_1 \wedge c_2) \vee (c_3 \wedge c_4) \leq (c_1 \vee c_3) \wedge (c_2 \vee c_4)$. For a constraint e, the set of constraints for a variable V_t can be written as $\{V_t = f_i(V_{t\cdot 1}, \ldots, V_{t\cdot m_i}) : 1 \leq i \leq l\}$ and we have $\alpha(V_t, e) = \{f_1/m_1, \ldots, f_l/m_l\}(T_1, \ldots, T_m)$ where $m = \max(m_1, \ldots, m_l)$ from the definition of broad type union. Applying γ to this function produces

$$\gamma(V_t, \alpha(V_t, e)) = \left(\begin{array}{c}(V_t = f_1(V_{t\cdot 1}, \ldots, V_{t\cdot m_1}) \\ \vee \cdots \vee \\ (V_t = f_l(V_{t\cdot 1}, \ldots, V_{t\cdot m_l})\end{array}\right) \wedge \gamma(V_{t\cdot 1}, T_1) \wedge \cdots \wedge \wedge \gamma(V_{t\cdot m}, T_m)$$

and using the above properties of constraints, $e \leq \gamma(\alpha(e))$. □

With a type system we are only concerned with the type of constraints that may allow a query to succeed. The abstract interpretation for types collapses the four truth values discussed in chapter 4 into two states: possibly true and definitely false. The types describe the situations under which a predicate is possibly true.

Definition 6.1.9 A *type abstraction* with depth k is a mapping of predicates onto terms $p/n \rightarrow p(T_1, \ldots, T_n)$, where depth$(T_i) \leq k, 1 \leq i \leq n$. The domain of type abstractions is denoted by $\mathcal{TA}$.

Assuming that the program, P, can be guard-stratified into a set of sub-programs, $\{P_1, \ldots, P_m\}$, the Φ_P operator is replaced by an abstract Φ'_P operator. This abstract operator takes a set of definitions and proceeds to successively approximate the workings of the Φ_P operator:

Definition 6.1.10 The *variable projection function* $\Pi : \mathcal{T} \times \mathcal{B} \times \mathcal{V} \rightarrow \mathcal{B}$ is defined as $\Pi(t, T, V) = \bigwedge_{t \diamond s = V} T \diamond s$ if T contains V, and $\top$ otherwise.

Given a goal $G = G_1, \ldots, G_n$, where each $G_i = p_i(t_{i1}, \ldots, t_{im_i})$ a type abstraction A and a variable V, the *type model* of V, written as $\Lambda(G, A, V)$ is defined as $\Lambda(G, A, V) = \bigwedge_{1 \leq i \leq n} \Pi(G, A(p_i/m_i), V)$

$\Phi'_P : \mathcal{TA} \times \mathcal{TA} \rightarrow \mathcal{TA}$ is defined as $\Phi'_P(A, B)(p/n) = A(p/n)$ if $p/n \in dom(A)$, and $\Phi'_P(A, B)(p/n) = p(T_1, \ldots, T_n)$ if $p/n \in dom(B)$, where each T_i is given by
$\bigvee_{p(V_1,\ldots,V_n)}$ `:-` $_{G_j \ \% \ B_j} \Lambda((G_j, B_j), A \cup B, V_i)$

Example 6.1.1 An example of the operation of the Φ'_P for the program:

```
p([], As, Bs)  :-  true ?  As = 0, Bs = 0.
p([a | R], As, Bs)  :-  true ?  p(R, As1, Bs), plus(As1, 1, As).
p([b | R], As, Bs)  :-  true ?  p(R, As, Bs1), plus(Bs1, 1, Bs).

q(L, T)  :-  p(L, As, Bs)  -> plus(As, Bs, T).
q(L, T)  :-  true  -> T = 0.
```

This program is guard stratified into two levels: $\{\{\texttt{p/3}\}, \{\texttt{q/2}\}\}$. For the purposes of this example the base type lattice has a base type of $\mathbf{I}$ for the set of integers, and `plus/3` has an initial type of $plus(\mathbf{I}, \mathbf{I}, \mathbf{I})$. The depth of the type abstraction is set to 2.

The first stratification has the fixpoint calculation:

$$\Phi'_{P_1} \uparrow 0 = \{\texttt{p/3} \rightarrow p(\top, \top, \top)\}.$$
$$\Phi'_{P_1} \uparrow 1 = \{\texttt{p/3} \rightarrow p(\{[]/0, ./2\}(\{a/0, b/0\}(), \top), \mathbf{I}, \mathbf{I})\}.$$
$$\Phi'_{P_1} \uparrow 2 = \{\texttt{p/3} \rightarrow p(\{[]/0, ./2\}(\{a/0, b/0\}(), \{[]/0, ./2\}(\{a/0, b/0\}, \top)), \mathbf{I}, \mathbf{I})\}.$$
$$\Phi'_{P_1} \uparrow \omega = \Phi'_{P_1} \uparrow 2$$

The second stratification has the fixpoint calculation:

$$\Phi'_{P_2} \uparrow 0 = \{\texttt{q/2} \rightarrow q(\top, \top)\}.$$
$$\Phi'_{P_2} \uparrow 1 = \{\texttt{q/2} \rightarrow q(\top, \mathbf{I})\}.$$
$$\Phi'_{P_2} \uparrow \omega = \Phi'_{P_2} \uparrow 1$$

Indexing Predicates

The information gathered by the fixpoint calculations can be used to provide accurate indexing of clauses using the clause set indexing from chapter 5. Broad types are particularly useful for clause set indexing as they reflect the way functors are gathered together at each point. For indexing purposes, each clause is numbered, and the types for each variable in the guard are computed. These types can then be used to build an indexing tree for each argument.

Definition 6.1.11 Suppose we have a suitable type abstraction for a program A and a predicate `p/n` defined by a series of clauses $p(V_1, \ldots, V_n)$ `:-` $G_i \ \% \ B_i, 1 \leq i \leq m$. Each clause generates an *indexing expression* $p(T_{i1}, \ldots, T_{in}) \rightarrow i$ where each T_{ij} is given by $T_{ij} = \Lambda(G_i, A, V_j)$

The *explicit argument set* for a trace t and set of indexing expressions $\{P \rightarrow i\}$ is defined as

$$\bigcup \{\Psi(P \diamond t) : P \diamond t \text{ is not a base type}\}.$$

Given a trace t and explicit argument set E for a set of indexing expressions $\{P \rightarrow i\}$, the *candidate clause set* for some functor/arity pair $f/l \in E$ is $\{i : f/l \in \Psi(P \diamond t)\}$.

The *alternate clause set* for a trace t and set of indexing expressions $\{P \rightarrow i\}$ with explicit argument set E is defined as $\{i : \Psi(P \diamond t) - E \neq \emptyset\}$

The candidate clause set gives the set of clauses that can be explicitly indexed by functor/arity pairs. The alternate clause set gives the set of clauses that can be successful, but cannot be explicitly indexed. The candidate clause sets for the interesting traces of `p/3` in example 6.1.1 are:

$$\begin{array}{ll} 1 & \{[]/0 \rightarrow \{1\}, ./2 \rightarrow \{2,3\}\} \\ 1.1 & \{a/0 \rightarrow \{2\}, b/0 \rightarrow \{3\}\} \end{array}$$

The candidate clause sets for the trace 1 of `q/2` in example 6.1.1 is $\{[]/0 \rightarrow \{1,2\}, ./2 \rightarrow \{1,2\}\}$. The alternate clause set for same trace is $\{2\}$.

6.1.3 Determining Modes

Modes carry information about the degree of constraint needed to ensure that a predicate becomes determinate, or at least not deadlock. The system of mode inferencing described here is designed to allow the compiler to order atoms within a goal and to delay calls until the arguments to the call have become sufficiently instantiated.

Mode inferencing is somewhat more subtle than type inferencing. In the case of some wait-guarded and conditional-guarded predicates it is possible to use the failure of other clauses to deduce the necessary modes for a clause.

```
p(a, Y)  :-  true ?  Y = b.
p(b, Y)  :-  true ?  Y = c.
p(X, Y)  :-  true ?  Y = a.

q(a, Y)  :-  true -> Y = b.
q(b, Y)  :-  true -> Y = c.
q(X, Y)  :-  true -> Y = a.
```

The last clause of `p/2` can only be determinately promoted if both preceding clauses have failed. To fail the preceding clauses of `p/2` must have had the first argument bound to some variable, and therefore the last clause implicitly assumes a bound first variable in the case of determinism. In a conditional guarded predicate, such as `q/2`, each clause is dependent on the failure of the preceding clauses and therefore carries implicit mode information. The last clause of `q/2` is dependent on the failure of the previous two clauses, and must have X bound in order to proceed.

The AKL is only implicitly moded. To allow abstract interpretation of an AKL program's modes we need to introduce a suitable concrete domain.

Definition 6.1.12 The *moding* of a predicate $p(V_1, \ldots, V_n)$ is a set of triples of constraints $u = \{< \theta_i, \theta_w, \theta_o >\}$ such that if there is a computation

$$\textbf{and}\,(p(V_1, \ldots, V_n); \theta)_{\{V_1, \ldots, V_n\} \cup \text{vars}(\theta)} \Rightarrow^* \textbf{choice}\,(\ \ldots, \textbf{and}\,(;\sigma)_{\text{vars}(\sigma)}, \ldots\ )$$

the computation will complete with no nondeterminate promotions and with $\theta_i \wedge \theta_w \wedge \theta_o \leftrightarrow \sigma$ if $\theta \rightarrow \theta_i \wedge \theta_w$. The computation will complete with one or more nondeterminate promotions and with $\theta_i \wedge \theta_w \wedge \theta_o \leftrightarrow \sigma$ if $\theta \rightarrow \theta_i$, but not $\theta \rightarrow \theta_i \wedge \theta_w$. Otherwise, the computation will deadlock.

θ_i represents an input mode, such as a constraint in the guard of a commit- or conditional-guarded predicate. θ_w represents a writable mode, such as a constraint in the guard of a wait-guarded predicate. Writable modes indicate constraints that can be promoted during nondeterminate promotion but could in preference be externally satisfied. θ_o represents an output mode, the constraints that are imposed by the body of a clause.

Modings are ordered, with two modings of `p/n`, u and v having $u \leq v$ if for all $< \theta_i, \theta_w, \theta_o > \in u$ there exists some $< \sigma_i, \sigma_w, \sigma_o > \in v$ such that $\theta_i \rightarrow \sigma_i$, $\theta_i \wedge \theta_w \rightarrow \sigma_i \wedge \sigma_w$ and $\theta_i \wedge \theta_w \wedge \theta_o \rightarrow \sigma_i \wedge \sigma_w \wedge \sigma_o$. Intuitively $u \leq v$ if u is a stricter moding than v.

The domain of modings is denoted by $\mathcal{M}$.

$m \wedge n$

		n			
		i	w	o	v
	i	i	w	o	i
	w	w	w	o	w
m	o	o	o	o	o
	v	i	w	o	v

$m \vee n$

		n			
		i	w	o	v
	i	i	w	o	v
	w	w	w	o	v
m	o	o	o	o	v
	v	v	v	v	v

$m \triangleright n$

		n			
		i	w	o	v
	i	i	i	i	i
	w	w	w	w	w
m	o	o	o	o	o
	v	i	w	o	v

Table 6.1: Abstract Mode Operators

An example mode for `q/2`, defined above, is $\{< X = a, \text{true}, Y = b >, < X = b, \text{true}, Y = c >, < X \neq a \wedge X \neq b, \text{true}, Y = a >\}$ The initial constraint $X = z$ will cause the computation to complete with the final constraint of $X = z \wedge Y = a$. However an initial constraint of true will deadlock, as true $\not\rightarrow X = a$, true $\not\rightarrow X = b$ and true $\not\rightarrow X \neq a \wedge X \neq b$.

The abstract domain is intended to model these mode partitions, allowing the construction of strong or weak delays in the abstract machine.

Definition 6.1.13 An *mode symbol* is one of $\mathcal{MS} = \{i, w, o, v\}$, with $i < w < o < v$.

A *mode tree* is either the mode v or $m(M_1, \ldots, M_n)$ where m is one of i, w or o and $M_1, \ldots, M_n$ are mode trees.

Mode trees have a similar definition of the depth function to broad types (see definition 6.1.4) and can be decomposed by the application of traces (see definition 6.1.2). The mode function is defined as $\text{mode}(v) = v$ and $\text{mode}(m(M_1, \ldots, M_n)) = m$.

Mode trees are partially ordered by $m \sqsubseteq v$ or $m(M_1, \ldots, M_n) \sqsubseteq l(L_1, \ldots, L_n)$, where $m \leq l$ and each $M_i \sqsubseteq L_i, 1 \leq i \leq n$.

A *mode abstraction* is a mapping from predicates $p(V_1, \ldots, V_n)$ to mode tree definitions of the form $p(M_1, \ldots, M_n)$.

The domain of mode trees is denoted by $\mathcal{MT}$ and that of mode abstractions is denoted by $\mathcal{MA}$.

Informally, *i* stands for input, *w* for writable, *o* for output and *v* for unconstrained variable. The abstraction is intended to model variables which are constrained by θ_i by *i*, θ_w by *w* and θ_o by *o*. Unconstrained variables are modelled by *v*.

To allow the modelling of operations on modes, mode trees have the following join and union operators. Generally input modes are overwritten by any output modes that have been produced.

Definition 6.1.14 The *join* of two mode symbols m and n, written as $m \wedge n$, the *union*, written as $m \vee n$, and the *tie*, written as $m \triangleright n$, are defined in table 6.1.

The *join* of two mode trees M and N, written as $M \wedge N$ is defined as

$$
\begin{array}{ll}
M & \text{if } N = v \\
N & \text{if } M = v \\
(m \wedge n)(M_1 \wedge N_1, \ldots, M_l \wedge N_l, N_{l+1}, \ldots, N_k) & \text{if } M = m(M_1, \ldots, M_l), \\
 & N = n(N_1, \ldots, N_k) \\
 & \text{and } l \leq k \\
(m \wedge n)(M_1 \wedge N_1, \ldots, M_k \wedge N_k, M_{k+1}, \ldots, M_l) & \text{if } M = m(M_1, \ldots, M_l), \\
 & N = n(N_1, \ldots, N_k) \\
 & \text{and } l \geq k
\end{array}
$$

The *union* of two mode trees M and N, written as $M \vee N$ is defined in a similar manner to the join, replacing the join of the mode symbols with the union. Similarly the *tie* of two mode trees, written as $M \triangleright N$ is defined by using the tie operator instead of the join operator.

The above mode tree representation does not allow the direct representation of negation. Mode trees implicitly represent a conjunction of constraints, and the negation operation converts a conjunction into a

disjunction, eg. $\neg X = f(a) \leftrightarrow X \neq f(\_) \vee (X = f(Y) \wedge Y \neq a)$. Similarly negations of constraints with multiple variables tends to separate the variables during negation, eg. $\neg(X = a \wedge Y = b) \leftrightarrow X \neq a \vee Y \neq b$. However it would be useful to be able to represent some of the effects of negation on modes. In these cases, the representation of negated modes must be reduced to an inaccurate approximation of the negation.

Definition 6.1.15 The *inversion* of a mode tree M, written as $-M$ is defined as

$$\begin{array}{ll} v & \text{if } M = v \\ m(v, \ldots, v) & \text{if } M = m(M_1, \ldots, M_n) \end{array}$$

We are now in a position to define suitable abstraction and concretisation functions for moving between predicate mode trees and modings.

Definition 6.1.16 The *concretisation function* $\gamma : \mathcal{MA} \rightarrow \mathcal{M}$ is defined as

$$\gamma(P) = \bigotimes_{\text{mode}(P \diamond t) \neq v} \gamma(V_t, P \diamond t)$$

where $< \theta_i, \theta_w, \theta_o > \otimes < \sigma_i, \sigma_w, \sigma_o > = < \theta_i \wedge \sigma_i, \theta_w \wedge \sigma_w, \theta_o \wedge \sigma_o >$ and $\gamma : \mathcal{V} \times \mathcal{MT} \rightarrow \mathcal{M}$ is defined as

$$\gamma(V, M) = \begin{cases} < \text{true}, \text{true}, \text{true} > & \text{if mode}(M) = v \\ < nonvar(V), \text{true}, \text{true} > & \text{if mode}(M) = i \\ < \text{true}, nonvar(V), \text{true} > & \text{if mode}(M) = w \\ < \text{true}, \text{true}, nonvar(V) > & \text{if mode}(M) = o \end{cases}$$

where $nonvar(V_t)$ is a shorthand for $\bigvee V_t = t(V_{t \cdot 1}, \ldots, V_{t \cdot n})$ for all functor/arity pairs and constants in the constraint domain; ie. $nonvar(V_t)$ requires V_t to be constrained to be some non-variable term.

As an example, $\gamma(p(i(v, w()))) = < nonvar(V_1), nonvar(V_{1.2}), \text{true} >$.

Definition 6.1.17 The *abstraction function* $\alpha : \mathcal{M} \rightarrow \mathcal{MA}$ is defined by

$$\alpha(u)(p(V_1, \ldots, V_n)) \diamond t = \bigvee_{\theta \in u(p(V_1, \ldots, V_n))} \alpha(V_t, \theta)$$

where $\alpha : \mathcal{V} \times \mathcal{M} \rightarrow \mathcal{MS}$ is defined as

$$\alpha(V, < \theta_i, \theta_w, \theta_o >) = \begin{cases} i & \text{if } \theta_i \text{ constrains } V \\ w & \text{if } \theta_w \text{ constrains } V \\ o & \text{if } \theta_o \text{ constrains } V \\ v & \text{otherwise} \end{cases}$$

An example abstraction of $p(V_1, V_2) \rightarrow \{< V_1 = a, \text{true}, V_2 = b >, < V_1 = b, \text{true}, V_2 = a >\}$ is $p(i, o)$.

Lemma 6.1.3 The functions γ and α as defined in definitions 6.1.16 and 6.1.17 are adjoined.

Proof The proof is an induction on the depth of the mode tree in a similar manner to the proof of lemma 6.1.2. Since variables are separated during abstraction, we need only consider individual predicate arguments and predicates of one variable.

If we have a mode tree M with depth$(M) = 1$ then $M = v$. In this case $\gamma(v) = <$ true, true, true $>$ and $\alpha(<$ true, true, true $>) = v$, so $\alpha(\gamma(M)) = M$, and $\gamma(\alpha(<$ true, true, true $>)) = <$ true, true, true $>$.

Suppose that the proposition holds for mode trees of depth $< n$. In such a case, we have $M = m(M_1, \ldots, M_n)$. If $m = i$ then $\gamma(p(M)) = < nonvar(V_1), \text{true}, \text{true} > \otimes \gamma(V_{11}, M_1) \otimes \cdots \otimes \gamma(V_{1n}, M_n)$. Since $\gamma(V_{11}, M_1), \ldots, \gamma(V_{1n}, M_n)$ do not constrain V_1, the final constraints will all be of the form $< nonvar(V_1) \wedge \theta_i, \theta_w, \theta_o >$ and $\alpha(V_1, <$ nonvar$(V_1) \wedge \theta_i, \theta_w, \theta_o >) = i$. By the induction hypothesis, this relationship holds true for $M_1, \ldots, M_n$ and therefore $\alpha(\gamma(p(M)) = p(M)$. A similar argument holds for the mode symbols w and o.

Suppose all modings of a predicate $< \theta_i, \theta_w, \theta_o >$ have θ_i constraining V_1. Then each $\alpha(V_1, < \theta_i, \theta_w, \theta_o >) = i$ and the union of all the mode symbols is i. In such a case, $\alpha(p(V_1)) = p(i(M_1, \ldots, M_n))$ and $\gamma(p(M)) =< nonvar(V_1), \text{true}, \text{true} > \otimes\gamma(V_{11}, M_1) \otimes \cdots \otimes \gamma(V_{1n}, M_n)$. Since each $\theta_i \rightarrow nonvar(V_1)$ we have $u(p(V_1) \leq \gamma(\alpha(u)p(V_1))$, since the induction hypothesis holds for each $M_1, \ldots, M_n$. A similar argument holds for the mode symbols w and o. □

With a pair of suitably adjoined domains, it is now possible to define an approximate Φ_P function for modes. This Φ'_P function must be able to approximate the effects of guards on the modes that are produced. In particular the Φ'_P operator must preserve the effects of guards. For example in the program

```
p(X)  :-  X = a  -> X = a.
```

`p/1` should have the mode $p(i)$, despite the implicit mode of o in the body. The tie operator can be used to preserve guard modes. We also need to be able to make modes more restrictive.

Definition 6.1.18 The *mode restriction operator* for a mode tree M and mode symbol n, written as $M \uparrow n$ is defined as

$$M \uparrow n = \begin{cases} v & \text{if } M = v \\ (\min(n, m))(M_1 \uparrow n, \ldots, M_l \uparrow n) & \text{if } M = m(M_1, \ldots, M_l) \end{cases}$$

We can now define a suitable approximation to the Φ_P operator.

Definition 6.1.19 The *variable projection function* $\Pi : \mathcal{T} \times \mathcal{MT} \times \mathcal{V} \rightarrow \mathcal{MT}$ is defined as $\Pi(t, M, V) = \bigwedge_{t \diamond s = V} M \diamond s$ if t contains V and v otherwise.

Given a goal $G = G_1, \ldots, G_n$, where each $G_i = p_i(t_{i1}, \ldots, t_{im_i})$, a mode abstraction u and a variable V, the *mode* of V, written as $\Lambda(G, A, V)$ is defined to be $\Lambda(G, A, V) = \bigwedge_{1 \leq i \leq n} \Pi(G, A(p_i/m_i), V)$

$\Phi'_P : \mathcal{MA} \times \mathcal{MA} \rightarrow \mathcal{MA}$ is defined as

1. $\Phi'_P(A, B)(p(V_1, \ldots, V_n) = A(p(V_1, \ldots, V_n))$ if $p(V_1, \ldots, V_n) \in dom(A)$.

2. $\Phi'_P(A, B)(p(V_1, \ldots, V_n)) = p(M_1, \ldots, M_n)$ if $p(V_1, \ldots, V_n) \in dom(B)$ and $p(V_1, \ldots, V_n)$ `:-` G_i `?` $B_i \in P, 1 \leq i \leq m$, where each $M_j = \bigvee_{1 \leq i \leq m}(\Lambda(G_i, B, V_j) \wedge -\Lambda(G_1, B, V_j) \wedge \cdots \wedge -\Lambda(G_{i-1}, B, V_j) \wedge -\Lambda(G_{i+1}, B, V_j) \wedge \cdots \wedge -\Lambda(G_m, B, V_j)) \uparrow w \triangleright \Lambda(B_i, B, V_j)$. Each $G_1, \ldots, G_{i-1}, G_{i+1}, \ldots G_m$ may only restrict one head variable.

3. $\Phi'_P(A, B)(p(V_1, \ldots, V_n)) = p(M_1, \ldots, M_n)$ if $p(V_1, \ldots, V_n) \in dom(B)$ and $p(V_1, \ldots, V_n)$ `:-` G_i `->` $B_i \in P, 1 \leq i \leq m$, where each $M_j = \bigvee_{1 \leq i \leq m}(\Lambda(G_i, B, V_j) \wedge -\Lambda(G_1, B, V_j) \wedge \cdots \wedge -\Lambda(G_{i-1}, B, V_j)) \uparrow i \triangleright \Lambda(B_i, B, V_j)$. Each $G_1, \ldots, G_{i-1}$ may only restrict one head variable.

4. $\Phi'_P(A, B)(p(V_1, \ldots, V_n)) = p(M_1, \ldots, M_n)$ if $p(V_1, \ldots, V_n) \in dom(B)$ and $p(V_1, \ldots, V_n)$ `:-` G_i `%` $B_i \in P, 1 \leq i \leq m$, where each $M_j = \bigvee_{1 \leq i \leq m}(\Lambda(G_i, B, V_j)) \uparrow i \triangleright \Lambda(B_i, B, V_j)$.

Example 6.1.2 This example of the mode Φ'_P operator is on the program

```
p([], X)  :-  true | X = 0.
p([_ | R], X)  :-  true | p(R, X1), plus(X1, 1, X).

q(L, X)  :-  p(L, X1)  ?  X = X1.
q(L, X)  :-  L = infinite  ?  X = -1.
```

This program is guard-stratified into $\{\{\texttt{p/2}\}, \{\texttt{q/2}\}\}$. Assuming a depth limit of 3 and that `plus/3` has the mode $plus(o, o, o)$, the fixpoint calculation for the first stratification is:

$$\Phi'_{P_1} \uparrow 0 = \{p(V_1, V_2) \rightarrow p(v, v)\}$$
$$\Phi'_{P_1} \uparrow 1 = \{p(V_1, V_2) \rightarrow p(i(v, v), o)\}$$
$$\Phi'_{P_1} \uparrow 2 = \{p(V_1, V_2) \rightarrow p(i(v, i(v, v)), o)\}$$
$$\Phi'_{P_1} \uparrow \omega = \Phi'_{P_1} \uparrow 2$$

The fixpoint calculation for the second stratification is:

$$\Phi'_{P_2} \uparrow 0 = \{q(V_1, V_2) \rightarrow p(v, v)\}$$
$$\Phi'_{P_2} \uparrow 1 = \{q(V_1, V_2) \rightarrow q(w(v, i(v, v)), o)\}$$
$$\Phi'_{P_2} \uparrow \omega = \Phi'_{P_2} \uparrow 1$$

Generating Delays

Mode abstraction has two uses. The first and most obvious use is to provide information so that suitable delay instructions can be constructed during the indexing of the predicate. If, for some trace t, mode$(p(M_1, \ldots, M_n) \diamond t) = i$, a strict delay can be added for the t argument to the indexing code for `p/n`. If mode$(p(M_1, \ldots, M_n) \diamond t) = w$ then a weak delay can be used for the t argument.

Ordering Goals

The second use of mode information is to allow the compiler to order goals within clauses to provide an optimal flow of bindings. The AKL assumes no particular order of execution in a list of atoms, beyond that of guards completing before bodies are executed. The compiler is therefore free to re-order atoms within goals to maximise execution efficiency. Efficiency is improved whenever moded predicates are called with bindings already in place, allowing an immediate commit to some clause.

The DAM is designed for use on parallel systems with a small number of processors, and these processors will rapidly be given work. After all processors have been assigned work, each processor will execute any goals sequentially. In general the normal mode of execution for any list of goals is a sequential first to last pattern. The modes available to the compiler from the abstract interpretation allow the goals of the sequence to be simply ordered.

To allow ordering, we assume an initial sequence of goals $G = G_1, \ldots, G_n$ and that there is a mode abstraction A for the program. For each pair of goals G_i, G_j we define $G_i \leq G_j$ if there exists a variable, $V \in$ vars(G) and traces t in G_i and s in G_j where mode$(A(G_i) \diamond t) \leq$ mode$(A(G_j) \diamond s)$ for all $V \in$ vars$(G_i) \cap$ vars(G_j) where $G_i \diamond t = G_j \diamond s = V$. If there is a cycle then, by the above definition, $G_i \leq G_j$ and $G_j \leq G_i$. In the case of a cycle, the original order of the goals is maintained, on the assumption that the programmer has some reason for ordering the goals in that way. Goal ordering is then a simple matter of topologically sorting the goals into descending order.

As an example, consider the goal `p(X, Y), q(Y, X), r(X)` where the mode abstraction is $A = \{p(V_1, V_2) \rightarrow p(i(), o()), q(V_1, V_2) \rightarrow q(i(), i()), r(V_1) \rightarrow r(o()), \}$

In this case we have $p(X, Y) \leq r(X)$ as both $p(X, Y) \diamond 1$ and $r(X) \diamond 1$ is X, and mode$(A(p(X, Y)) \diamond 1) = i$, mode$(A(r(X)) \diamond 1) = o$. We also have $q(X, Y) \leq r(X)$. For the above example, $q(X, Y) \leq p(Y, X)$, since mode$(A(p(X, Y)) \diamond 2) = o$, mode$(A(p(Y, X)) \diamond 1) = i$.

Sorting the goals gives the sequence `r(X), p(X, Y), q(Y, X)`.

6.2 Compilation on Partial Information

The advantages of logic programming languages for compilers have been recognised since [War80]. Most compilation can be viewed as a meta-interpretation of the source code, evaluating it to the point that it can be transformed into another target language. Since the source and target languages are unlikely to have an exact mapping between statements, this approach tends to become muddied by collecting forward information about the interpretation.

Logic programming languages allow a more direct approach to compiler writing by using logical variables and difference lists. The use of logical variables means that exact decisions about some elements, such as register allocations, can be deferred until enough information is available. At that point it is possible to "fill in" the elements associated with the logical variables before continuing. The use of difference lists allows compilation to proceed as an interpretation of the code, with the various parts of the code being assembled at the end.

The AKLs determinacy condition can be used to make a compiler even closer to the ideal of a linear code interpreter[1]. Decisions can be deferred until enough data becomes available by allowing each point in the compilation to combine data from the "past" — pre-built terms, permanent register allocations, etc — with data from the "future" — specific register positions, modes, etc. Provided that the compiler can continue with the meta-interpretation, the deferrable parts of the compilation can be spawned, to await sufficient information. This view of AKL compilation is also used in the AGENTs compiler supplied by SICS.

6.2.1 Temporary Register Allocation

Since the DAM is a register-based abstract machine, each term in the computation will need to be allocated a register for its lifetime within the computation. An optimal register allocation minimises the amount of moves between pairs of registers and between registers and memory; the optimal allocation eliminates unnecessary "register shuffling".

Since the DAM has 256 temporary registers it is unlikely that temporary terms will have to be moved from and to temporary storage between calls. The DAM uses a register passing convention for passing arguments to calls; an ideal register allocation would ensure that terms which are used as arguments to calls are always pre-placed.

Generating an optimal register allocation is known to be an NP-hard problem [MJT91]. However register allocation for the DAM essentially needs to satisfy only two criteria: a term remains in a register while it is in use and a term is placed in a calling register if it is to be used in the next call.

Register allocation in the DAM compiler consists of two streams. The forward stream consists of register allocations made in the computation's past, where terms are matched against logical variables. The backward stream consists of allocation requests from the future, where a register is requested as having 'any' register, or a specific register number. When a term stops being requested by the future, the term can be discarded from the set of register allocations, allowing the register to be re-allocated. The two stream approach combines both optimal register placement and liveness analysis.

Following Matyska et al. [MJT91], we use the following terminology. An *inline* call is a call to a primitive inline predicate, eg. unification or arithmetic operations. An *out of line* call is a call to any other predicate. A *chunk* is a sequence of zero or more inline calls, terminated by the end of the clause, a guard operator or an out of line call.

The domain of registers is denoted by $\mathcal{R} = \{0 \ldots m\}$, where m is the maximum register number — 255 in the case of the DAM.

Definition 6.2.1 A *register assignment* is a function $A : \mathcal{T} \rightarrow \wp\mathcal{R}$ which maps any term onto a set of temporary registers; the registers that currently hold a reference to the term. $\mathcal{A}$ represents the domain of register assignments.

The *used register set* for a register assignment A is defined as

$$Used(A) = \cup\{A(t) : t \in dom(A)\}$$

Definition 6.2.2 A *liveness set* is a function $L : \mathcal{T} \rightarrow \wp\mathcal{R} \cup \{\bot, \top\}$ which maps any term onto the registers that could best be used to hold the term; on to $\bot$ if the term is no longer live, or onto $\top$ to indicate a "don't care" assignment. For the purposes of set union and intersection $\bot \equiv \{\}$ and $\top \equiv \mathcal{R}$. $\mathcal{L}$ represents the domain of liveness sets.

The *unavailable register set* for a liveness set L and term t is defined as

$$Unavail(L, t) = \cup\{L(s) : s \in dom(L) \text{ and } s \neq t\}$$

[1] At least in concept; actual execution is liable to become very intricate.

Definition 6.2.3 The *clash set* between an assignment A and liveness set L is the function $Clash : \mathcal{A}\times\mathcal{L} \rightarrow \wp\mathcal{T}$ where

$$Clash(A, L) = \{t : L(t) \neq \bot \text{ and } A(t) \cap L(t) = \emptyset\}$$

Definition 6.2.4 An *assignment function* is a function $Ass : \mathcal{P} \times \mathcal{A} \times \mathcal{L} \rightarrow \mathcal{A} \times \mathcal{L}$.

Given a chunk c with predicates $p_1, \ldots, p_n$ and some assignment function, we have a corresponding sequence of register assignments and liveness sets $(A_1, L_1), \ldots, (A_{n+1}, L_{n+1})$ where A_1 is some initial register assignment, $L_{n+1} = \{t \rightarrow \bot : t \in \mathcal{T}\}$ and $Ass(p_i, A_i, L_i) = (A_{i+1}, L_{i+1})$. Assuming that all register transfers, both permanent and temporary, take up a similar amount of time,[2] the cost of a particular assignment function Ass for the chunk c is given by

$$Cost(Ass, c) = \Sigma_{i=1}^{n+1} |Clash(A_i, L_i)|$$

An optimal assignment function for c minimises $Cost(Ass, c)$.

Given a predicate $p_i(t_{i1}, \ldots, t_{im})$ and letting $T = \{t_{i1}, \ldots, t_{im}\}$, the register assignment function for the DAM uses the following elements:

If p_i is an inline predicate then

$$L_i = \begin{array}{l} \{t \rightarrow L_{i+1}(t) : t \notin T\}\cup \\ \{t \rightarrow L_{i+1}(t) : L_{i+1}(t) \neq \bot\}\cup \\ \{t \rightarrow \top : L_{i+1}(t) = \bot \wedge t \in T\} \end{array}$$

If p_i is an out of line predicate then

$$L_i = \begin{array}{l} \{t \rightarrow L_{i+1}(t) : t \notin T\}\cup \\ \{t \rightarrow \{j : t = t_{ij}\} : L_{i+1}(t) = \top\}\cup \\ \{t \rightarrow L_{i+1}(t) \cup \{j : t = t_{ij}\} : L_{i+1}(t) \neq \top\} \end{array}$$

The register assignment for the DAM is computed as follows:

If p_i is an inline predicate, then $A_{i+1} = A_i^m$ where

$$A_i^j = \begin{array}{c} \{t \rightarrow A_i^{j-1}(t) - \{Ra(t_{ij}, A_i^{j-1}, L_i\} : t \neq t_{ij}\}\cup \\ \{t \rightarrow A_i(t) \cup \{Ra(t_{ij}, A_i^{j-1}, L_i)\} : t = t_{ij}\} \end{array}$$

$Ra(t, A, L)$ is defined to be the lowest element of the first non-empty set in the sequence: A, $L(t) - Unavail(L, t)$, $\mathcal{R} - Unavail(L, t)$ or $\mathcal{R}$ and $A_i^0 = A_i$.

If p_i is an out of line predicate, then $A_{i+1} = A_i^m$ where

$$A_{i+1} = \begin{array}{c} \{t \rightarrow A_i(t) - 1, \ldots, m : t \notin T\}\cup \\ \{t \rightarrow (A_i(t) - \{k : t \neq t_{ik}, 1 \leq k \leq m\}) \cup \{j\} : t = t_{ij}\} \end{array}$$

As an example, consider the chunk `A = f(X), p(A, A)` where $A_1 = \{X \rightarrow \{1\}\}$ (all terms not included in the set are assumed to be $t \rightarrow \emptyset$). The sequence of liveness sets is $L_3 = \{\}, L_2 = \{A \rightarrow \{1, 2\}\}, L_1 = \{A \rightarrow \{1, 2\}, X \rightarrow \top\}$. The sequence of register assignments is $A_1 = \{X \rightarrow \{1\}\}, A_2 = \{X \rightarrow \{1\}, A \rightarrow \{2\}\}, A_3 = \{A \rightarrow \{1, 2\}\}$.

Once a register has been assigned, appropriate instructions can be inserted into the instruction stream to allow the construction of the term in the register.

[2] In the DAM, temporary and permanent registers are stored as vectors in memory, so this assumption is justified. More complex machines, with different transfer costs, need a more complex weighting

		Processors			
Benchmark	Compiler	1	2	2	4
nrev(500)	Full	1900	1300	900	900
	No Modes	2000	1500	1000	1100
	No Indexing	15000	13000	13000	24000
bad-qsort(1000)	Full	1700	1300	1200	1200
	No Goal-Ordering	3500	3000	2800	2900
and(10000)	Full	1600	2700	4700	7000
	No Clause Sets	5300	6300	7800	8700
	No Indexing	6200	7300	8400	8800
All results in ms, $\pm 10\%$.					

Table 6.2: Performance of Selectively Compiled Code

6.2.2 Permanent Register Allocation

Permanent register assignment can follow the same pattern as temporary register assignment. Requests for terms needed in the future can be passed back to the origin of the term and a permanent register assigned to the term on the construction of the term. Register assignment occurs during the last use of the term in the permanent register, allowing re-use of permanent registers. A permanent register assignment is needed wherever a term is used in two chunks.

Following section 6.2.1, we use the previous definitions of register assignment, liveness set and register assignment function (definitions 6.2.1, 6.2.2 and 6.2.4). The register assignment function for permanent register assignment, is given for a predicate $p(t_1, \ldots, t_n), T = \{t_1, \ldots, t_n\}$ by:

$$L_i = \{L_{i+1}(t) : t \notin T\} \cup \{t \rightarrow \{\min(\mathcal{R} - Unavail(L_{i+1}, t))\} : t \in T\}$$

If p is an in-line predicate, then $A_{i+1} = A_i$
If p is an out of line predicate, then

$$A_{i+1} = \begin{array}{l} \{t \rightarrow \emptyset : L_i(t) = \perp\} \cup \\ \{A_i(t) : t \notin T \text{ and } L_i(t) \neq \perp\} \cup \\ \{t \rightarrow Ra(t, A_i, L_i) : t \in T \text{ and } L_i(t) \neq \perp\} \end{array}$$

As an example, the permanent register assignment for the clause `?-p(X) :- q(1, Y), r(X), s(Y))` would have the liveness set sequence $L_1 = \{Y \rightarrow \{1\}, X \rightarrow \{2\}\}$, $L_2 = \{Y \rightarrow \{1\}, X \rightarrow \{2\}\}$, $L_3 = \{Y \rightarrow \{1\}\}$ and $L_4 = \emptyset$ and the register assignment sequence $A_1 = \{X \rightarrow \{2\}\}$, $A_2 = \{X \rightarrow \{2\}, Y \rightarrow \{1\}\}$, $A_3 = \{Y \rightarrow \{1\}\}$, $A_4 = \emptyset$.

6.3 Performance

The effects on performance that the abstract interpretations and optimisations have can be measured by selectively crippling the compiler and seeing what effect that has on the speed of compiled programs.

Three benchmarks were run with parts of the compiler removed. **nrev(500)** is the naive reverse benchmark described in section 5.5. **bad-qsort(1000)** is the quicksort benchmark described in section 5.5, but with the goals in the clauses reversed, to create an inefficient goal ordering. **and(10000)** is the and operator benchmark described in section 5.5. Only those benchmarks where the compiler can introduce additional efficiency have been included in the compiler results. Smaller versions of these benchmarks were used, as the DAM tended to run into memory problems while executing the crippled versions of the full benchmarks. The results are summarised in table 6.2.

On the naive reverse benchmark, the removal of moding code does not affect the performance of the benchmark appreciably; although consistent, most improvements are within the bounds of error. This result is unsurprising, as the calls to the reverse and append predicates are likely to have fully instantiated arguments when called. When run in parallel, the suspensions still create choice-boxes, the most expensive part

of the computation, leading to only a minor improvement in performance. Removal of indexing produces a spectacular drop in performance, as numerous choice- and and-boxes are created and speculative bindings made. With 4 processors, the performance is almost double the sequential performance; if a call with bound variables is made, then the call can quickly commit to the correct clause, if a call with unbound variables is made, then the call must suspend and then be re-woken when a constraint fails.

Goal-ordering improves the performance of the bad quicksort program so that it runs as well as a sensibly ordered program (which is what has been produced by the goal ordering). The overhead associated with the non-goal ordered program is largely the creation of choice-boxes (again).

Using clause sets provides a very clear performance improvement. Without clause sets, most calls to `and/3` will create at least two and-boxes. With clause sets, a determinate call can be detected and a clause committed to before any and box is created.

Chapter 7

Conclusions

The benchmarks taken for the DAM show that, sequentially at least, the DAM is an efficient abstract machine for use with the AKL. However the expense of some operations and granularity problems make its parallel performance less than optimal. In addition, the selection rule of the DAM tends to encourage broad exploration for solutions, rather than a narrow, focused search for an initial solution.

In hindsight, there are a large number of implementation decisions that I would change if I were to re-implement the DAM. Clause sets are an obvious success, and I think that localised variables are a nice way of looking at the problem of multiple constraints on a variable, albeit more memory expensive than suspensions or hash windows. Other than these two elements, the DAM could do with a re-design:

- To be truly useful, the nondeterministic behaviour of the DAM will need to be modified. The DAM essentially schedules on a first-come, first-served basis, with work being added to the work queue as it appears. To control nondeterminism, a scheduler will need to both restrict nondeterministic promotion until all determinate work has been processed and ensure that the remaining choices in a nondeterministically promoted choice-box are left until the branch that has been promoted has been fully explored. The DAM should still be able to explore branches in or-parallel if the workers are available.

  A possible solution to this problem is a scheduling algorithm which maintains a queue of and-parallel work, a stack of available nondeterminate promotions and a stack of branches split from nondeterminate promotions. Work can be preferentially allocated from the and-parallel queue, the nondeterminate promotion stack and the branch stack in that order. By preferentially exploring a single branch during nondeterminate promotion, the search is narrowed. However idle workers will still be able to acquire work from the nondeterminate promotion and branch stacks if no and-parallel work is available.

- Granularity remains a significant problem. Ideally, idle workers should only acquire box with a suitable amount of further work to be done. However it is not easy to pre-estimate the amount of work that a box contains when deciding whether to queue the work. It is possible that work can be divided into two classes: light-weight work, such as woken and committed boxes, and small nondeterminate promotions and heavy-weight work, such as parallel calls and large nondeterministic promotions. Light work can be performed in-line by the worker that detects the work. Heavy work can be queued.

  Abstract interpretation may be able to detect likely cases of large or small amounts of work, and suitably annotate clauses with simple conditional statements in a manner similar to CGEs. An example of this approach would be an expression that limits parallel calls on the Fibonacci benchmark to those with input arguments of (eg.) 4 or more, preventing work fragmentation [DLH90].

- The copying approach to nondeterminism seems to be the simplest approach to implementing the AKL. Some way of introducing backtracking would improve nondeterminate performance, however.

- When I started to design the DAM, I was influenced by the RISC approach to architecture, hence the splitting of the get and unification instructions into separate testing, locking and binding instructions. Since these instructions tend to be simple and not take very much time to execute, the overhead associated with decoding them tends to dominate. For example, most variables are local, so the Localise instruction normally does nothing. The splitting of testing and structure creation tends to create a code explosion whenever complex terms are unified. A re-implementation of the DAM would return to the traditional WAM-style get and unify instructions, although I am still concerned about the possibility of two processors, one reading and one writing the same term getting into a race condition.
- Variables, particularly unconstrained variables are too big. The Penny approach, where unconstrained variables can be represented by a single cell is very much superior.
- Box creation is too expensive. Part of the expense of box creation is the linking of boxes together and the locking overheads that this entails. If only a single processor can access a box, some way of avoiding locking is needed. A scheme of shallow backtracking, where and- and choice- boxes are lazily created would also help.
- The DAM uses a unified heap. Separating the heap into a heap, a box stack and an environment stack would allow easier memory reclamation.

The most interesting aspect of this work on the AKL is probably the fixpoint semantics developed in chapter 4. This semantics was originally developed to allow a suitable semantics for the bottom-up abstract interpretation of the AKL, a language with elements of committed-choice, and the logical difficulties that that introduces. By extending Boolean logic to the bilattice model and providing semantics in this logic for commit operators, a coherent logical semantics can be built for the AKL and for other, simpler committed-choice languages.

A missing feature in the fixpoint semantics for the AKL is that it has no concept of deadlock. Extending the semantics by including the moded forms given in chapter 6 would provide a more accurate semantics.

Appendix A

Benchmark Code

This appendix contains the code for the various benchmarks used to test the system.

A.1 nrev(1000)

```
generate(0, L) :- true -> L = [].
generate(N, L) :- true -> L = [N | L1], N1 is N - 1, generate(N1, L1).

nrev([], L) :- true ?  L = [].
nrev([X | Rest], Rev) :- true ?  nrev(Rest, Rev1), append(Rev1, [X], Rev).

append([], I, O) :- true ?  I = O.
append([X | Rest], I, O) :- true ?  O = [X | ORest], append(Rest, I, ORest).

?- generate(1000, L), nrev(L, _L2).
```

A.2 qsort(2500)

```
generate(0, _S, L) :- true -> L = [].
generate(N, S, L) :-
       true
       ->
       L = [S | L1],
       S1 is ((S * 2311) + 25637) mod 4081,
       N1 is N - 1,
       generate(N1, S1, L1).

qsort([], T, S) :- true ?  T = S.
qsort([P | U], T, S) :-
       true
       ?
       partition(P, U, UL, UR),
       qsort(UL, T, SL),
       qsort(UR, [P | SL], S).

partition(_P, [], L, R) :-
       true
       ?
       L = [],
```

```
       R = [].
partition(P, [X | U], L, R) :-
       X >= P
       ?
       L = [X | L1],
       partition(P, U, L1, R).
partition(P, [X | U], L, R) :-
       X < P
       ?
       R = [X | R1],
       partition(P, U, L, R1).

?- generate(2500, 10, L), qsort(L, [], _L2)
```

A.3 fib(25)

```
fib(0, FN) :- true -> FN = 1.
fib(1, FN) :- true -> FN = 1.
fib(N, FN) :-
       true
       ->
       N1 is N - 1,
       fib(N1, FN1),
       N2 is N - 2,
       fib(N2, FN2),
       add(FN1, FN2, FN).

% ?- mode explicitly adds mode information to a predicate
?- mode add(i, i, o).
add(A, B, C) :- C is A + B.

?- fib(25, _).
```

A.4 tree(17)

```
tree(0, T) :- true -> T = nil.
tree(N, T) :-
       true
       ->
       T = t(N, T1, T2),
       N1 is N - 1,
       tree(N1, T1),
       tree(N1, T2).

?- tree(17, _).
```

A.5 subset(15)

```
% Style used by NU-Prolog and DAM
sstest(X) :- subset(X, _S) ? fail.
sstest(_X) :- true ?  true.
```

```
% Style used by AGENTS and Penny
sstest(X) :- unordered_bagof(R, subset(X, R), _L).

subset([], S) :- true ?  S = [].
subset([X | X1], [X | S1]) :- true ?  subset(X1, S1).
subset([_X | X1], S) :- true ?  subset(X1, S).

?- sstest([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15])
```

A.6 encap(7)

```
% Style used by NU-Prolog and DAM
encap :- xprod([1, 2, 3, 4, 5, 6, 7], _R) ? fail.
encap.

% Style used by AGENTS and Penny
encap :- unordered_bagof(R, xprod([1, 2, 3, 4, 5, 6, 7], R), _L).

xprod(X, S1-S2) :- xprod1(X, S1), xprod1(X, S2).

xprod1(X, S) :- subset(X, S1) ?  S = S1.

subset([], S) :- true ?  S = [].
subset([X | X1], [X | S1]) :- true ?  subset(X1, S1).
subset([_X | X1], S) :- true ?  subset(X1, S).
```

A.7 filter(1000)

```
filter :-
       allowed(1000, A),
       generate(1000, L),
       filter(L, A, _L1).

allowed(N, L) :- N =< 0 -> L = [].
allowed(N, L) :- true ->
       L = [N | L1],
       N1 is N - 10,
       allowed(N1, L1).

generate(0, L) :- true -> L = [].
generate(N, L) :- true ->
       L = [N | L1],
       N1 is N - 1,
       generate(N1, L1).

filter([], _A, F) :- true -> F = [].
filter([L | LR], A, F) :-
       memberchk(L, A)
       ->
       F = [L | FR],
       filter(LR, A, FR).
```

```
filter([_L1 | LR], A, F) :- true ->
       filter(LR, A, F).

memberchk(A, [A | _R]) :- true -> true.
memberchk(A, [_A | R]) :- true -> memberchk(A, R).
```

A.8 and(50000)

```
andtest :- andtest1(50000).

andtest1(0).
andtest1(N) :- N > 0 ?
       and(1, 0, X),
       and(1, Y, X),
       and(_, 1, Y),
       N1 is N - 1,
       andtest1(N1).

and(0, 0, 0) :- true ?  true.
and(0, 1, 0) :- true ?  true.
and(1, 0, 0) :- true ?  true.
and(1, 1, 1) :- true ?  true.
```

Appendix B

Sample DAM Code

This appendix contains the compiled code for `partition/4` in the quicksort benchmark given in section A.2.

```
        PRED        'partition'/4

0:      SwitchOnTerm     0,&1,&2,Fail,Fail
1:      ChoiceBox        4
        Suspend          0,1,&0
2:      SwitchOnTerm     1,&3,&11,&4,Fail
3:      ChoiceBox        4
        Suspend          1,1,&2
4:      GetListArgument  1,1,5
        SwitchOnTerm     5,&5,&6,&6,&6
5:      ChoiceBox        4
        Suspend          5,1,&4
6:      ChoiceBox        4
        Try              0,0,&21
        Try              0,0,&31
        Defer

11:     TryOne
12:     GetConstant      1,$'[]',&13
13:     GetConstant      2,$'[]',&14
        Lock             2,&13
        Localise         2,&13
        BindConstant     2,$'[]'
14:     GetConstant      3,$'[]',&15
        Lock             3,&14
        Localise         3,&14
        BindConstant     3,$'[]'
15:     Proceed

21:     AndBox
        Allocate         4
        GetListArgument  1,1,4
        LessEq           4,0,Fail
        PutValueXY       0,0
        PutValueXY       1,1
        PutValueXY       2,2
```

```
        PutValueXY      3,3
        WaitCommit
        Promote
        Raise
        PutValueYX      2,2
22:     GetList         2,&23
        Lock            2,&22
        Localise        2,&22
        PutList         3
        PutValueYX      1,1
        GetListArgument 1,1,4
        PutListArgument 4,1,3
        PutVariableX    4
        PutListArgument 4,2,3
        BindVariable    3,2
        Jump            &25
23:     GetListArgument 2,1,3
        PutValueYX      1,1
        GetListArgument 1,1,4
        GetVariable     4,3
25:     PutValueYX      0,0
        PutValueYX      1,1
        GetListArgument 1,2,1
        PutValueYX      2,2
        GetListArgument 2,2,2
        PutValueYX      3,3
        Deallocate      4
        Execute         'partition'/4

31:     AndBox
        Allocate        4
        GetListArgument 1,1,4
        Less            0,4,Fail
        PutValueXY      0,0
        PutValueXY      1,1
        PutValueXY      2,2
        PutValueXY      3,3
        WaitCommit
        Promote
        Raise
        PutValueYX      3,3
32:     GetList         3,&33
        Lock            3,&32
        Localise        3,&32
        PutList         4
        PutValueYX      1,1
        GetListArgument 1,1,5
        PutListArgument 5,1,4
        PutVariableX    5
        PutListArgument 5,2,4
        BindVariable    4,3
        Jump            &35
33:     GetListArgument 3,1,4
        PutValueYX      1,1
```

```
        GetListArgument  1,1,5
        GetVariable      5,4
35:     PutValueYX       0,0
        PutValueYX       1,1
        GetListArgument  1,2,1
        PutValueYX       2,2
        PutValueYX       3,3
        GetListArgument  3,2,3
        Deallocate       4
        Execute          'partition'/4

        LAST
```

Appendix C

Abbreviations

AKL Andorra Kernel Language or Agents Kernel Language

BAM Basic Andorra Model

CCL Committed-Choice Language

CGE Conditional Graph Expression

DAP Dependent And-Parallelism

DAM Doug's Abstract Machine

DDAS Dynamic Dependent And-Parallel Scheme

DDM Data Diffusion Machine

GHC Guarded Horn Clauses

IAP Independent And-Parallelism

JAM Jim's Abstract Machine

KAP Kernel Andorra Prolog

KL1 Kernel Language 1

MIMD Multiple Instruction Multiple Data

SIMD Single Instruction Multiple Data

WAM Warren Abstract Machine

Bibliography

[AK90] Khayri A. M. Ali and Roland Karlsson. The Muse Or-Parallel Prolog model and its performance. In Saumya Debray and Manuel Hermenegildo, editors, *Proceedings of the 1990 North American Conference on Logic Programming*, pages 757–776, Austin, 1990. ALP, MIT Press.

[AK91a] Hassan Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, 1991.

[AK91b] Khayri A. M. Ali and Roland Karlsson. Scheduling Or-parallelism in Muse. In Koichi Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 807–821, Paris, France, 1991. The MIT Press.

[Ali86] Khayri A. M. Ali. Or-parallel execution of prolog on a mutli-sequential machine. *International Journal of Parallel Programming*, 15(3):189–214, June 1986.

[AM88] H. Alshawi and D. B. Moran. The delphi model and some preliminary experiments. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1578–1589, Seatle, 1988. ALP, IEEE, The MIT Press.

[Bah91] Reem Bahgat. *Pandora: Non-Deterministic Parallel Logic Programming*. PhD thesis, Imperial College, 1991.

[BDL+88] Ralph Butler, Terry Disz, Ewing Lusk, Robert Olson, Ross Overbeek, and Rick Stevens. Scheduling OR-Parallelism: An Argonne perspective. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1590–1605, Seatle, 1988. ALP, IEEE, The MIT Press.

[Bel77] N. D. Belnap. A useful four-valued logic. In J. Michael Dunn and G. Epstein, editors, *Modern Uses of Multiple-Valued Logic*, pages 8–37. Reidel, 1977.

[BG89] R. Bahgat and S. Gregory. Pandora: Non-deterministic parallel logic programming. In Giorgio Levi and Maurizio Martelli, editors, *Proceedings of the Sixth International Conference on Logic Programming*, pages 471–486, Lisbon, 1989. The MIT Press.

[BJ88] Maurice Bruynooghe and Gerda Janssens. An instance of abstract interpretation integrating type and mode inferencing. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 669–683, Seatle, 1988. ALP, IEEE, The MIT Press.

[Bor84] P. Borgwardt. Parallel Prolog using stack segments on shared-memory multiprocessors. In *Proc. International Symposium on Logic Programming*, pages 2–11, Atlantic City, 1984. IEEE, Computer Society Press.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, January 1977.

[CG86] Keith L. Clark and Steve Gregory. Parlog: parallel programming in logic. *ACM Transactions on Programming Languages and Systems*, 8(1):1–49, January 1986.

[Cha85] Jung-Heng Chang. High performance execution of logic programs based on a static data dependency analysis. Technical Report UCB/CSD 86/263, Univerity of California at Berkeley, 1985.

[Cod91] Michael Codish. *Abstract Interpretation of Sequential and Concurrent Logic Programs*. PhD thesis, Weizmann Institute of Science, January 1991.

[Con83] John S. Conery. *The AND/OR process model for parallel interpretation of logic programs*. Ph.D. thesis, Department of Information and Computer Science, University of California, Irvine, June 1983.

[Con92] John S. Conery. The opal machine. In Peter Kacsuk and Michael J. Wise, editors, *Implementations of Distributed Prolog*, pages 159–185. Wiley, 1992.

[Cra88] Jim Crammond. *Implementation of Committed Choice Languages on Shared Memory Multiprocessors*. PhD thesis, Department of Computing, Imperial College of Science and Technology, London, England, May 1988.

[CRR92] Ta Chen, I. V. Ramakrishnan, and R. Ramesh. Multistage indexing algorithms for speeding Prolog execution. In Krzysztof Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 639–653, Washington, USA, 1992. The MIT Press.

[CS89] A. Calderwood and P. Szeredi. Scheduling or-parallelism in Aurora: The Manchester scheduler. In Giorgio Levi and Maurizio Martelli, editors, *Proceedings of the Sixth International Conference on Logic Programming*, pages 419–435, Lisbon, 1989. The MIT Press.

[DeG84] Doug DeGroot. Restricted and-parallelism. In *Proceedings of the 1984 International Conference on Fifth Generation Computer Systems*, pages 471–478, Tokyo, Japan, November 1984.

[DLH90] Saumya K Debray, Nai-Wei Lin, and Manuel Hermenegildo. Task granularity analysis in logic programs. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–188, June 1990.

[FBR93] Steve Frank, Henry Burkhadt, and James Rothric. The KSR1: Bridging the gap between shared memory and mpps. In *Proceedings of COMPCON*, pages 285–294, Spring 1993.

[FF92] Sz. Ferenczi and I. Futó. CS-Prolog: A communicating sequential prolog. In Peter Kacsuk and Michael J. Wise, editors, *Implementations of Distributed Prolog*, pages 357–378. Wiley, 1992.

[Fit91] Melvin Fitting. Bilattices and the semantics of logic programming. *The Journal of Logic Programming*, 11(1 & 2):91–116, July 1991.

[Fly66] M. J. Flynn. Very high speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, December 1966.

[Foo94] Wai-Keong Foong. A directory-based scheme of implementing distributed shared memory for multi-transputer systems. In *Australasian Workshop on Parallel and Real-Time Systems*, pages 135–149. Victoria University of Technology, 1994.

[Fra94] Torkel Franzén. Some formal aspects of AKL. Research Report R94:10, Swedish Institute of Computer Science, Kista, Sweden, 1994.

[FW91] Ian Foster and Will Winsborough. Copy avoidance through compile-time analysis and local reuse. In Vijay Saraswat and Kazunori Ueda, editors, *Logic Programming, Proceedings of the 1991 International Symposium*, pages 455–469, San Diego, USA, 1991. The MIT Press.

[GH91] Gopal Gupta and M. Hermenegildo. Ace: And/or-parallel copying-based execution of logic programs. In Anthony Beaumont and Gopal Gupta, editors, *Proceedings of the ICLP91 Pre-Conference Workshop on Parallel Execution of Logic Programs*, Paris, France, June 1991.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractibility*. W.H. Freeman, San Francisco, 1979.

[GJ89] G. Gupta and B. Jayaraman. Combined And-Or Parallism on Shared Memory Multiprocessors. In Ewing L. Lusk and Ross A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming*, pages 332–349, Cleveland, Ohio, USA, 1989.

[GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1081–1086, Seatle, 1988. ALP, IEEE, The MIT Press.

[GL92] Maurizio Gabbrielli and Giorgio Levi. Unfolding and fixpoint semantics of concurrent constraint logic programs. *Theoretical Computer Science*, 105(1):85–128, 1992.

[GMS96] Laura Giordano, Alberto Martelli, and Maria Luisa Sapino. Extending negation as failure by abduction: A three valued stable model semantics. *The Journal of Logic Programming*, 26(1):31–68, 1996.

[Got87] A. Goto. Parallel inference machine research in FGCS project. In *Proceedings of the US-Japan AI Symposium 87*, pages 21–36, 1987.

[Han92] Werner Hans. A complete indexing scheme for WAM-based abstract machines. Technical Report TR92-11, RWTH Aachen, Lehrstuhl für Informatik II, 1992.

[HB88] Seif Haridi and Per Brand. Andorra Prolog — an integration of Prolog and committed choice languages. In *Proceedings of the 1988 International Conference on Fifth Generation Computer Systems*, pages 745–754, Tokyo, Japan, December 1988.

[Her86a] Manuel V. Hermenegildo. An abstract machine for restricted AND-parallel execution of logic programs. In Ehud Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, Lecture Notes in Computer Science, pages 25–39, London, 1986. Springer-Verlag.

[Her86b] Manuel V. Hermenegildo. Efficient management of backtracking in AND-Parallelism. In Ehud Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, Lecture Notes in Computer Science, pages 40–54, London, 1986. Springer-Verlag.

[HG91] Arie Harsat and Ran Ginosar. CARMEL-4: The unify-spawn machine for FCP. In Koichi Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 840–854, Paris, France, 1991. The MIT Press.

[HJ90] Seif Haridi and Sverker Janson. Kernel Andorra Prolog and its computation model. In David H. D. Warren and Peter Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 31–46, Jerusalem, 1990. The MIT Press.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[Jan94] Sverker Janson. *AKL: A Multiparadigm Programming Language.* PhD thesis, Uppsala University, 1994.

[JH91] Sverker Janson and Seif Haridi. Programming paradigms of the Andorra Kernel Language. In Vijay Saraswat and Kazunori Ueda, editors, *Logic Programming, Proceedings of the 1991 International Symposium*, pages 167–186, San Diego, USA, 1991. The MIT Press.

[JL87] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the Fourteenth Conference on the Princliples of Programming Languages*, pages 111–119. ACM Press, 1987.

[JLW90] Dean Jacobs, Anno Langen, and Will Winsborough. Multiple specialization of logic programs with run-time test. In David H. D. Warren and Peter Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 717–731, Jerusalem, 1990. The MIT Press.

[Kac92] Peter Kacsuk. Distributed data driven Prolog abstract machine. In Peter Kacsuk and Michael J. Wise, editors, *Implementations of Distributed Prolog*, pages 89–118. Wiley, 1992.

[Kal87] L. V. Kalé. The REDUCE-OR process model for parallel evaluation of logic programs. In Jean-Louis Lassez, editor, *Proceedings of the Fourth International Conference on Logic Pro gramming*, MIT Press Series in Logic Programming, pages 616–632, Melbourne, 1987. The MIT Press.

[Kow74] R. Kowalski. Predicate logic as programming language. In Jack L. Rosenfeld, editor, *Proceedings of the Sixth IFIP Congress (Information Processing 74)*, pages 569–574, Stockholm, Sweden, August 1974.

[KS88] Shmuel Kliger and Ehud Shapiro. A decision tree compilation algorithm for FCP (|, :, ?). In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1315–1336, Seatle, 1988. ALP, IEEE, The MIT Press.

[KT91] M. Korsloot and E. Tick. Compilation techniques for nondeterminate flat concurrent logic programming languages. In Koichi Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 457–471, Paris, France, 1991. The MIT Press.

[LBD+90] Ewing Lusk, Ralph Butler, Terrence Disz, Robert Olson, Ross Overbeek, Rick Stevens, D.H.D. Warren, Alan Calderwood, Peter Szeridi, Seif Haridi, Per Brand, Mats Carlsson, Andrzej Ciepielewski, and Bogumil Hausman. The Aurora or-parallel Prolog system. *New Generation Computing*, 7(2, 3):243–271, 1990.

[LK88] Y. J. Lin and V. Kumar. And-parallel execution of logic programs on a shared memory multiprocessor: A summary of results. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1123–1141, Seatle, 1988. ALP, IEEE, The MIT Press.

[Llo84] John W. Lloyd. *Foundations of Logic Programming*. Springer series in symbolic computation. Springer-Verlag, New York, 1984.

[LM92] Thomas LeBlanc and Evangelos Markatos. Shared memory vs. message passing in shared memory multiprocessors. Technical report, University of Rochester, Rochester, NY 14627, April 1992.

[MA96] Johan Montelius and Khari A. M. Ali. An and/or-parallel implementation of AKL. *New Generation Computing*, 14(1):31–52, 1996.

[MD93] Remco Moolenaar and Bart Demoen. A parallel implementation for AKL. In *Programming Language Implementation and Logic Programming: PLILP'93*, pages 246–261, 1993.

[MD94] Remco Moolenaar and Bart Demoen. Hybrid tree search in the Andorra model. In Pascal Van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 110–123. MIT Press, 1994.

[MH90] K. Muthukumar and M. V. Hermenegildo. The DCG, UDG, and MEL methods for automatic compile-time parallelization of logic programs for independent and-parallelism. In David H. D. Warren and Peter Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 221–236, Jerusalem, 1990. The MIT Press.

[MH92] K. Muthukumar and M. Hermenegildo. Compile-time derivation of variable dependency using abstract interpretation. *The Journal of Logic Programming*, 13(1, 2, 3 and 4):315–347, 1992.

[Mil91] Håkan Millroth. Reforming compilation of logic programs. In Vijay Saraswat and Kazunori Ueda, editors, *Logic Programming, Proceedings of the 1991 International Symposium*, pages 485–502, San Diego, USA, 1991. The MIT Press.

[MJT91] L. Matyska, A. Jergová, and D. Toman. Register allocation in WAM. In Koichi Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 142–156, Paris, France, 1991. The MIT Press.

[MNL88] K. Marriott, L. Naish, and J.-L. Lassez. Most specific logic programs. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 909–923, Seatle, 1988. ALP, IEEE, The MIT Press.

[Mon97] Johan Montelius. *Exploiting Fine-grain Parallelism in Concurrent Constraint Languages*. PhD thesis, Uppsala University, Uppsala, Sweden, April 1997.

[MS92] Kim Marriott and Harald Søndergaard. Bottom-up dataflow analysis of normal logic programs. *The Journal of Logic Programming*, 13(1, 2, 3 and 4):181–204, 1992.

[MS93] Maged Michael and Michael Scott. Fast mutual exclusion, even with contention. Technical report, University of Rochester, Rochester, NY 14627, June 1993.

[MWB90] Anne Mulkers, William Winsborough, and Maurice Bruynooghe. Analysis of shared data structures for compile-time garbage. In David H. D. Warren and Peter Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 747–762, Jerusalem, 1990. The MIT Press.

[Nai86] Lee Naish. *Negation and control in Prolog*. Number 238 in Lecture Notes in Computer Science. Springer-Verlag, New York, 1986.

[Nai88] Lee Naish. Parallelizing NU-Prolog. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1546–1564, Seatle, 1988. ALP, IEEE, The MIT Press.

[Nai89] Lee Naish. Proving properties of committed choice logic programs. *The Journal of Logic Programming*, 7(1):63–84, July 1989.

[Nai93] Lee Naish. Applying the Andorra principle. *Australian Computer Science Communications*, 15(1):191–201, 1993.

[Nak92] K. Nakajima. Distributed implementation of kl1 on the multi-psi. In Peter Kacsuk and Michael J. Wise, editors, *Implementations of Distributed Prolog*, pages 89–118. Wiley, 1992.

[PN84] Luis Pereira and Roger Nasr. Delta Prolog: A distributed logic programming language. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 283–291, Tokyo, Japan, November 1984.

[PN91] Doug Palmer and Lee Naish. NUA-Prolog: An extension to the WAM for parallel Andorra. In Koichi Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 429–442, Paris, France, 1991. The MIT Press.

[RDC92] S Raina, Warren D. H. D., and J. Cownie. Parallel prolog on a scalable multiprocessor. In Peter Kacsuk and Michael J. Wise, editors, *Implementations of Distributed Prolog*, pages 27–44. Wiley, 1992.

[Rou75] P. Roussel. Prolog: Manuel de référence et d'utilisation. Technical report, Université d'Aix-Marseille, Groupe d'Intelligence Artificielle, 1975.

[Sar87] Vijay A. Saraswat. The concurrent logic programming language CP: definition and operational semantics. In *Conference Record of the Fourteenth ACM Symposium on Principles of Programming Languages*, pages 49–62, Munich, West Germany, January 1987.

[SC93] Vítor Santos Costa. *Compile-Time Analysis for the Parallel Execution of Logic Programs in Andorra-I*. PhD thesis, University of Bristol, Bristol, UK, August 1993.

[Sch95] Erik Schön. On the computation of fixpoints in static program analysis with an application of analysis of AKL. Technical Report R95:06, Swedish Institute of Computer Science, October 1995.

[SCWY91a] Vítor Santos Costa, David H. D. Warren, and Rong Yang. The Andorra-I engine: A parallel implementation of the basic Andorra model. In Koichi Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 825–839, Paris, France, 1991. The MIT Press.

[SCWY91b] Vítor Santos Costa, David H. D. Warren, and Rong Yang. The Andorra-I preprocessor: Supporting full Prolog on the basic Andorra model. In Koichi Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 443–456, Paris, France, 1991. The MIT Press.

[Sha83] Ehud Y. Shapiro. A subset of Concurrent Prolog and its interpreter. Technical Report ICOT TR-003, Institute for New Generation Computer Technology, Tokyo, Japan, 1983.

[Sha86] E. Shapiro. Concurrent Prolog: A progress report. In *Fundamentals of Artificial Intelligence*, number lncs 232, pages 277–313. Springer-Verlag, 1986.

[She92] Kish Shen. Exploiting dependent and-parallelism in Prolog. The dynamic dependent and-parallel scheme (DDAS). In Krzysztof Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 717–731, Washington, USA, 1992. The MIT Press.

[She93] Kish Shen. Implementing dynamic dependent and-parallelism. In David S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 167–183. MIT Press, June 1993.

[SIC88] SICS. Sicstus Prolog user's manual. Technical Report R88007B, Swedish Institute of Computer Science, 1988.

[Som87] Z. Somogyi. A system of precise models for logic programs. In Jean-Louis Lassez, editor, *Proceedings of the Fourth International Conference on Logic Pro gramming*, MIT Press Series in Logic Programming, pages 769–787, Melbourne, 1987. The MIT Press.

[Som89] Zoltan Somogyi. *A parallel logic programming system based on strong and precise modes*. PhD thesis, Department of Computer Science, University of Melbourne, Melbourne, Australia, January 1989. Technical Report 89/4.

[SRV88] Zoltan Somogyi, Kotagiri Ramamohanarao, and Jayen Vaghani. A stream and-parallel execution algorithm with backtracking. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 1142–1159, Seatle, 1988. ALP, IEEE, The MIT Press.

[SS95] Thomas Sjöland and Dan Sahlin. Fixpoint analysis of type and alias in AKL programs. Technical Report R94:13b, Swedish Institute of Computer Science, January 1995.

[Sta80] T. A. Standish. *Data Structure Techniques*. Addison-Wesley, 1980.

[Tic95] Evan Tick. The deevolution of concurrent logic programming languages. *JLP*, 23(1, 2, 3):89–124, 1995.

[UC90] K. Ueda and T. Chikayama. Design of the kernel language for the parallel inference machine. *The Computer Journal*, 33(6):494–500, 1990.

[Ued86] Kazunori Ueda. *Guarded Horn Clauses*. PhD thesis, University of Tokyo, Tokyo, Japan, March 1986.

[UF88] K. Ueda and K Furukawa. Transformation rules for ghc programs. In ICOT, editor, *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 582–591, ICOT, Japan, December 1988.

[War80] David H. D. Warren. Logic programming and compiler writing. *Software Practise and Experience*, 10(2):97–125, 1980.

[War83] David H.D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, Menlo Park, California, October 1983.

[War87] D.H.D. Warren. The SRI model for or-parallel execution of Prolog: Abstract design and implementation. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 92–102, San Francisco, August - September 1987. IEEE, Computer Society Press.

[WH88] David H. D. Warren and Seif Haridi. The data diffusion machine: A scalable shared virtual memory multiprocessor. In ICOT, editor, *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 943–952, ICOT, Japan, December 1988.

[Win92] Will Winsborough. Multiple specialization using minimal-function graph semantics. *The Journal of Logic Programming*, 13(1, 2, 3 and 4):259–290, 1992.

[WR87] H. Westphal and P. Robert. The PEPSys model: Combining backtracking, AND- and OR-parallelism. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 436–448, San Francisco, August - September 1987. IEEE, Computer Society Press.

[XG88] H. Xia and W.K. Giloi. A hybrid scheme for detecting and-parallelism in Prolog programs. In *International Conference on Supercomputing*, pages 539–559. ACM Press, July 1988.

[YA87] Rong Yang and Hideo Aiso. P-Prolog: a parallel logic language based on exclusive relation. *New Generation Computing*, 5(1):79–95, 1987.

[YKS90] E. Yardeni, S. Kliger, and E. Shapiro. The languages FCP(:,?) and FCP(:). *New Generation Computing*, 7(2, 3):89–107, 1990.

[Zob90] Justin Zobel. *Types in Logic Programming*. PhD thesis, University of Melbourne, 1990.

[ZT86] Justin Zobel and James Thom. NU-Prolog reference manual, version 1.0. Technical Report 86/10, Department of Computer Science, University of Melbourne, Melbourne, Australia, 1986.

[ZT91] Kang Zhang and Ray Thomas. A non-shared binding scheme for parallel Prolog implementation. In John Mylopoulos and Ray Reiter, editors, *Proceedings of the 12th International Joint Conference on Artifical Intelligence*, pages 877–882. Morgan Kaufmann Publishers, Inc., August 1991.

Index