# Determinacy and Determinacy Analysis


Pat Hill*
School of Computer Studies,
University of Leeds,
Leeds, LS2 9JT, UK.
hill@scs.leeds.ac.uk

Andy King†
Computing Laboratory,
University of Kent at Canterbury,
Canterbury, CT2 7NF, UK.
a.m.king@ukc.ac.uk


December 19, 1995


**Abstract**

One unique feature of logic languages, which makes them well-suited for artificial intelligence problems, is their ability to succinctly and declaratively express non-determinacy and hence search. Improving search effiency is one of the main goals of AI and, by studying how redundant search may be factored out, this paper contributes to this goal. In logic programming, alternatives can be specified by a set of sentences defining the same predicate. By backtracking, considering in turn each of these sentences, these alternatives can be explored until a solution (if one exists) is found. However, though backtracking is essential for certain parts of a program, typically, many predicates are deterministic, and most queries to a program have no more than one solution. Providing for non-determinacy can slow down the execution of a program on a uni-processor and limit the scope for parallel execution on a multi-processor. As a consequence, programmers are often forced to resort to the non-logical features of the language to ensure any determinacy is fully exploited. A number of papers on determinacy and its detection have been published. However, because of the diversity of applications for determinacy analysis, there has been a similar diversity of definitions of determinacy and its related concepts. This paper reformulates the determinacy definitions in a uniform framework; identifying and contrasting the different approaches. Techniques for detecting and exploiting determinacy are also reviewed together with some directions for future research.


## 1 Introduction

In logic programming, alternatives can be specified by a set of sentences defining the same predicate. By backtracking, considering in turn each of these sentences, these alternatives can be explored until a solution (if one exists) is found. However, though backtracking is essential for certain parts of a program, typically, many predicates are deterministic, and most queries to a program have no more than one solution. Providing for non-determinacy can slow down the execution of a program on a uni-processor and limit the scope for parallel execution on a multi-processor.

In the sequential execution of a program, non-deterministic code will normally cause the creation of choice-points and these are computationally expensive in both time and memory. As all


*Supported by EPSRC Grant GR/H/79862.
†Supported, in part, by ESPRIT project 6707 "ParForce".

implementations, including parallel ones, have sequential components, it is clear that detecting deterministic code at compile time can reduce the number of unnecessary choice points and improve the efficiency of the resulting code. In practice, programmers often resort to the non-logical features of the language such as the Prolog cut to ensure any determinacy is fully exploited. However, programs with features such as the cut are difficult to write and expensive to maintain. Moreover, as concluded in [3], for inductive logic programming with current induction techniques, it is not practical to learn logic programs with the cut. It would be better if a more declarative program was written (or learnt) and then either annotated with determinacy information or the deterministic parts detected in an automatic way.

The efficient distribution of work to as many of the available processors as possible is fundamental to any parallel implementation. In broad terms, there are two forms of parallelism that have been recognised, *or-parallelism* and *and-parallelism*. Or-parallelism exploits the non-determinacy in a program by exploring the different parts of the search tree in parallel. However, or-parallelism is rarely scalable and there are often insufficient alternative paths in a program to make it viable. The other form of parallelism, called *and-parallelism*, computes the different conjuncts of a goal in parallel. In the latter case, the implementation is simplified if backtracking is local to a processor. This can be ensured by executing goals in parallel only if they do not share variables (independent and-parallelism), or, more generally, any shared variables are not re-assigned (dependent and-parallelism). One sufficient condition for the latter is that the predicate which binds the shared variable must be deterministic.

Determinacy information can be used by program transformers such as partial evaluators [24] and program verifiers in proofs of properties such as termination [2]. Normally, the intention of a program transformer is to create (possibly specialised) code that is more efficient than the original. Thus it is important that the transformation minimises the amount of non-determinacy in the transformed program. Although an important transformation technique is unfolding, unfolding non-deterministic code can cause an explosion in the number of clauses increasing the non-determinacy as well as repeating computations unnecessarily [24]. It is shown in [2] that, if a goal is determinate and has at least one SLD-derivation that terminates successfully, then every SLD-derivation for that goal will terminate successfully. This and similar results are particularly important for logic programming languages that have flexible computation rules or parallel implementations where the behaviour and hence termination is less predictable.

It is expected that a knowledge of determinacy would also be of benefit to theorem provers. Many analysis and transformation techniques that have been developed for logic programming can also be successfully applied to automatic theorem proving. For example, in [15], it has already been demonstrated that, by avoiding sentences in a theory that cannot contribute to a proof, significant improvements can be made to the efficiency of the theorem prover.

In spite of the importance of determinacy in providing efficient implementations, there is a lack of consensus as to its precise nature. There is the classic division in logic programming between the declarative and procedural view of a program. Under the declarative approach, determinacy, called here *functionality*, depends on the uniqueness of a correct answer for the goal and program. Under the procedural interpretation, computational behaviour is characterised by SLDNF-derivation. Here, it is useful to separate the structure of a derivation from the representation of its substitutions [7]. This helps to distinguish between the two main forms of procedural determinacy: the *structural* and the *substitutional* forms. In structural determinacy, a node in the search tree is required to have no more than one child node, while the substitutional form is based on the uniqueness of variable bindings. These are clearly distinct. For example, with the program

$P(A)$
$P(x) \leftarrow Q(x)$
$Q(A)$

where $A$ is a constant, the query $\leftarrow P(v)$ generates a unique binding for $v$ and therefore has one computed answer $v = A$. It is thus substitutionally deterministic. The query, however, has two child nodes in the search tree and therefore is not structurally deterministic. Note that the determinacy terminology, *functionality, structural determinacy,* and *substitutional determinacy* is defined for this paper and is introduced so that we can clearly distinguish these different forms of determinacy.

Prolog is inherently sequential and literals in the goal and clauses in the program are processed according to their textual ordering. The predominance of Prolog as a logic programming language has meant that most work on determinacy presupposes a leftmost computation rule and a depth-first search rule. For instance, the Prolog cut (which relies on the sequential computation procedure described above) has had a major influence on previous work on determinacy. This is because, by pruning alternative paths in the search tree, it is possible to remove choice points, thereby making the program more deterministic.

Because of the drive for exploiting parallelism, however, many recent innovative logic programming schemes such as NU-Prolog [48], Reform Prolog [4], Andorra [25], and Gödel [27] have been devised with a more flexible control procedures but allowing for explicit control directives by the programmer. Moreover, new pruning operators such as the *one solution* in NU-Prolog, the *commit* in Andorra, and the Gödel *commit*, that do not depend, like the traditional cut, on a fixed order of execution, have been designed and implemented. However, although more flexible control schemes have recently been considered with respect to data flow analysis ([17, 18]) and the role of determinacy has been explored in a partial way for some less restrictive control schemes, the more general and important problem of detecting determinacy for an arbitrary control strategy has never been formally defined let alone solved and implemented. In this paper, we formalise the concept of determinacy for flexible control strategies and therefore lay the foundation for future analysis and implementation work.

The rest of the paper is organised as follows. First, in Section 2, the terminology and background concepts used in the rest of the paper are presented. Sections 3, 4, and 5 describe and define each of the three forms of determinacy; functional, structural, and substitutional, as outlined above, comparing each with similar definitions in the literature. In Section 6, we examine the different techniques that have been used to detect determinacy in logic programs. Methods for exploiting determinacy information in the compiler to produce optimised code are discussed in Section 7. The paper concludes with a summary and some directions for further research.

## 2 Preliminaries

Before discussing the different notions of determinacy, some notation and preliminary definitions are required.

First, note that, in the example programs, names of constants, functions, and predicates have an upper-case letter as the first character whereas names of variables have a lower-case letter as the first character. An underscore denotes a unique variable existentially quantified in front of the atom in which it occurs. It is assumed that all other (free) variables are universally quantified at the left of the clause in which they occur. Prolog notation ($[]$, $[\ldots]$, $[\ldots|\ldots]$) is used for lists. The symbol $\Pi$ is reserved to denote an arbitrary program. Logic programming terminology used in this paper is based on the book [30]. Only where, for the purposes of this paper, this terminology is modified or extended is it defined here.

For any logic program there is an underlying language, either inferred from the symbols used in the program (as in Prolog) or defined by language declarations (as in Gödel). It is assumed here that this language always includes at least one constant so that the Herbrand universe is non-empty. Any practical programming language will provide *built-in predicates* which are either part of the underlying language or, in a modular language such as Gödel, provided by the system modules. By a *built-in* atom, we mean an atom whose predicate is built-in. In particular, it is assumed that the programming language has a built-in binary infix predicate = defined by the single clause

$x$=$x$.

The programs in this paper are assumed to be normal logic programs (where the body of the program clause or goal is a conjunction of literals). Also, unless otherwise specified, the programs are assumed to be *pure*; that is, free of cuts, commits, or any other non-declarative constructs. Pure logic programs permit flexible computation rules and facilitate co-routining and parallelisation. It is assumed that the declarative semantics of a logic program $\Pi$ is based on its completion comp($\Pi$) and that any implementation of the program is based on the procedural semantics of SLD-resolution and its extension to SLDNF-resolution for negative literals.

As the concepts of correct and computed answer are central to the definitions of functionality and substitutional determinacy, we repeat them here.

**Definition** Let $\Pi$ be a program and $\leftarrow W$ a goal. A *correct answer* for $\Pi \cup \{\leftarrow W\}$ is a substitution restricted to the variables in $W$ such that $\forall W\theta$ is a logical consequence of comp($\Pi$). A *computed answer* for $\Pi \cup \{\leftarrow W\}$ is the composition of substitutions used in an SLDNF-refutation for $\Pi \cup \{\leftarrow W\}$ restricted to the variables in $W$.

To simplify the paper, all the substitutions are assumed to be idempotent. Moreover, when defining determinacy in terms of the uniqueness of a correct or a computed answer, we do not wish to consider irrelevant differences in the names of variables in the range of the substitution.

**Definition** If $\theta, \theta'$ are substitutions, then we say $\theta, \theta'$ are *equivalent* if there exist substitutions $\sigma, \sigma'$ such that $\theta' = \theta\sigma$, $\theta = \theta'\sigma'$.

In order to correctly model the run-time behaviour of a program (in particular, backtracking) we adapt the definition of an SLDNF-tree in [30] to include, at each node, edges for both successful and failed unification tests. Thus we assume that each node, for which the selected node is an atom, has $k$ children; one for each clause in the definition of the predicate in the atom. Those children for which the unification test fails, are labelled with failure, while the remaining children are as in the standard definition of an SLDNF-tree. Note that as before, if there are no children, then the node is also labelled with failure.

Any implementation also has to consider how to search the SLDNF-tree for a solution. In particular if the determinacy is detected dynamically, then it will depend on the structure of the unexplored part of the tree. In searching an SLDNF-tree, once a node has been visited, usually a record is kept of any child nodes that have not yet been considered. Provided there is at least one such child, then the parent node is called a *choice point*.

In making a general study of determinacy it is important to try to separate control issues from the definitions since in some computational models the selection rule is defined in terms of the possible deterministic behaviour of the program. For example, in Andorra [25], determinate atoms are selected in preference to non-determinate ones. On the other hand, the control still needs to be considered since the order in which literals are selected can affect the determinacy. There are a number of ways in which the intended selection rules can be specified. In Prolog, the textual order of the literals determines the precise order of selection. In Gödel, control declarations, define restrictions on the atoms that may be selected from a goal. Thus, for generality, we assume for each program the existence of a function called a *selection rule* that maps goals to sets of literals.

**Definition** A *selection rule* $\mathcal{S}$ for a program $\Pi$ is a function that maps a goal $G$ for $\Pi$ to a subset of the literals in that goal[1] such that,

1. if $G$ is of the form $\leftarrow W_1 \wedge W_2$, then $\mathcal{S}(G) \subseteq \mathcal{S}(\leftarrow W_1) \cup \mathcal{S}(\leftarrow W_2)$;
2. if $\theta$ is any substitution, then $\mathcal{S}(G)\theta \subseteq \mathcal{S}(G\theta)$;
3. No non-ground negative literals occur in $\mathcal{S}(G)$.

Any literal in $\mathcal{S}(G)$ is said to be *selectable* in $G$ (wrt $\mathcal{S}$). If $G$ is non-empty and no literal is selectable, then we say that the goal (and the derivation in which it occurs) *flounders* (wrt $\mathcal{S}$). The *trivial* selection rule is one that maps any goal $G$ to the set consisting of all positive literals and ground negative literals in $G$.

Condition 1 says that the selection rule must distribute over conjunction. Moreover, if a literal $L$ is selectable in some goal, then it is selectable in the goal $\leftarrow L$. Condition 2 ensures that if a literal in a goal is selectable, then an instance of it will be selectable in any instance of that goal. Condition 3 is required for safe negation and ensures the soundness of the SLDNF-derivation procedure.

A selection rule may force sequentiality by choosing, for each goal, at most one literal. For example, the *left to right* rule of Prolog in which only the leftmost literal in the current goal may be selected is a sequential selection rule. Similarly, the *right to left* rule where only the rightmost literal may be selected is another sequential selection rule. A more flexible selection rule may be defined by mode or control declarations that only restrict the selectable literals. This may leave some non-determinacy in the choice of the next literal for resolution, allowing for parallel execution of the selectable literals. For example, in Gödel [27], the order of the literals does not affect the selection rule and the rule only has to observe the control declarations in the program and the built-in control. Some Prolog systems such as NU-Prolog [48] allow some variation of the left to right rule, with user-defined control declarations and safe negation partly determining the selection rule. Most Prolog systems have some control for the built-in predicates, requiring some of their arguments to be ground. Note that when referring to an SLDNF-tree or a computed answer, unless otherwise specified, the trivial selection rule is assumed.

It is useful to show that the determinacy definitions given later in this paper are well-behaved with respect to the selection rules. That is, if a selection rule is replaced by another which allows less literals to be selected, then any determinacy property should be preserved.

**Definition** Suppose there are two selection rules $\mathcal{S}$ and $\mathcal{S}'$ for a program $\Pi$ such that for any goal $G$ for $\Pi$,

$$\mathcal{S}(G) \subseteq \mathcal{S}'(G)$$

then we say that $\mathcal{S}$ is *stronger* than $\mathcal{S}'$.

[1] In [30], a computation rule is defined as a function from goals to literals

Given a program and selection rule for that program, there is, for each predicate in the program, a set of *modes* specifying the degree to which its arguments and subterms of these arguments are instantiated when an atom with that predicate is selected. In particular, if an argument (or a subterm of that argument) of the predicate has to be non-variable before the atom is selectable, then the mode of the position of that argument (or subterm) is said to be *input* and the position is said to be an *input position*. The modes of the predicates can be partly defined by means of control declarations (as in Gödel) or inferred from the built-in control and underlying selection rule (as in Prolog) [33].

# 3 Functionality

In declarative programming, the programmer states the logic of the problem without considering the method of computation and its implementation. Thus, the declarative form of determinacy, called here functionality, may only be understood in terms of the uniqueness of logically correct answers for a query. Functionality, has been shown to be important in database applications [34].

**Definition** Let $\Pi$ be a program with selection rule $\mathcal{S}$. Then a goal $G$ is *functional* if there is (up to equivalence) at most one correct answer for comp($\Pi \cup \{G\}$). A predicate $P/n$ is *functional* wrt $\mathcal{S}$ if every goal of the form $\leftarrow\ P(t_1,\ldots,t_n)$, where $P(t_1,\ldots,t_n)$ is selectable wrt $\mathcal{S}$, is functional.

As a consequence, any goal whose literals are either negative or atoms whose predicates are functional, will also be functional.

**Example 3.1** *Consider the program*

$P(x,y)\ \leftarrow\ Q(x,y)$
$Q(A,B)$
$Q(A,A)$.

*Then, the goal* $\leftarrow P(v,A)$ *is functional with correct answer* $\{v=A\}$. *The goal* $\leftarrow P(B,A)$ *is also functional as it has no correct answer. The goal* $\leftarrow\ P(A,v)$ *has two correct answers* $v=A$ *and* $v=B$, *so that it is not functional. More generally, if the selection rule is trivial, then* $P/2$ *is not functional. However, if* $P(t_1,t_2)$ *is only selectable when* $t_2$ *is ground, the predicate* $P/2$ *will be functional.* □

The next proposition states some basic properties of functionality.

**Proposition 3.1** *Let* $\Pi$ *be a program and* $\mathcal{S}$ *a selection rule for* $\Pi$. *Then,*

1. *if a predicate is functional wrt* $\mathcal{S}$, *it is functional wrt any stronger selection rule;*
2. *if a goal* $\leftarrow W$ *is functional wrt* $\mathcal{S}$, *the goal* $\leftarrow W\theta$ *is functional wrt* $\mathcal{S}$ *for any substitution* $\theta$;
3. *if the goals* $\leftarrow W$ *and* $\leftarrow W_1$ *are functional wrt* $\mathcal{S}$, *the goal* $\leftarrow W \wedge W_1$ *is functional wrt* $\mathcal{S}$.

□

**Proof** 1 is obvious. For 2, suppose that $\leftarrow W\theta$ is not functional, then any variant is also not functional and we can assume that $\theta$ is idempotent. Then there are at least two non-equivalent correct answers, $\phi_1$ and $\phi_2$ for comp($\Pi$) $\cup \{\leftarrow W\theta\}$. As $\phi_1$ and $\phi_2$ are restricted to variables in

$W\theta$, they do not bind variables in the domain of $\theta$. Thus $\theta\phi_1$ and $\theta\phi_2$ are non-equivalent correct answers for comp$(\Pi) \cup \{\leftarrow W\}$.

For 3, suppose that $\leftarrow W \wedge W_1$ is not functional and that $\phi_1$ and $\phi_2$ are non-equivalent correct answers for comp$(\Pi) \cup \{\leftarrow W \wedge W_1\}$. As there is at least one constant in the language, we can assume that $\phi_2$ is a ground substitution. As the order of the subgoals $W, W_1$ is not significant in the proof, we can assume that the restriction $\phi'_1$ and $\phi'_2$ of $\phi_1$ and $\phi_2$ to the variables in $W$ are distinct and that as $\phi'_2$ is a ground substitution, they are not equivalent. Then, as $\phi'_1$ and $\phi'_2$ are correct answers for comp$(\Pi) \cup \{\leftarrow W\}$, $\leftarrow W$ is not functional. $\Box$

As functionality expresses a declarative view of determinacy, it is straightforward for the user to state which predicates are functional. For example, in [34] (where the program representing the database is assumed to be function free) the user can include *functional dependency* statements of the form:

$$P(x_1, \ldots, x_n): \ \ x_{j_1}, \ldots, x_{j_p} \ \rightarrow \ x_j$$

(where $j \notin \{j_1, \ldots, j_p\}$). The $j$'th argument is said to *functionally depend* on the $j_1, \ldots, j_p$ arguments. This means that the correct answers for any two goals $\leftarrow P(s_1, \ldots, s_n)$ and $\leftarrow P(t_1, \ldots, t_n)$ with identical $j_k$'th arguments, $1 \leq k \leq p$, will agree on the binding for any variables in the $j$'th argument. Thus, if, for example, the $n$'th argument functionally depends on the $1, \ldots, n-1$ arguments and $P/n$ is only selectable when the first $n-1$ arguments are constants, $P/n$ will be functional. The following example taken from [34] illustrates the use of functional dependencies.

**Example 3.2** *Consider a database of employees defined by predicates BigProject/1, Salary/2, and Project/3 where the arguments to Project/3 denote, respectively, the project name, the manager of the project, and an employee in the project. The arguments to Salary/2 denote, respectively, an employee's name and salary.*

$$\mathit{BigProject}(p) \ \leftarrow \ \mathit{Project}(p, m, e) \ \wedge \ \mathit{Salary}(m, s) \ \wedge \ s > 51000$$

$\mathit{Project}(p, m, e): \ p \ \rightarrow \ m$
$\mathit{Project}(p, m, e): \ e \ \rightarrow \ p$
$\mathit{Project}(\mathit{Turing}, \mathit{Ric}, \mathit{Steve})$
$\mathit{Project}(\mathit{Turing}, \mathit{Ric}, \mathit{Jim})$
$\mathit{Salary}(\mathit{Ric}, 50000)$.

*Then, assuming that* $s > 51000$ *is only selectable when s is bound to a ground term, with the goal*

$$\leftarrow \ \mathit{BigProject}(\mathit{Turing})$$

*calls to Project/*3 *and Salary/*2 *will succeed with substitution* $\{m = \mathit{Ric}, e = \mathit{Steve}, s = 50000\}$. *The goal*

$$\leftarrow \ 50000 > 51000$$

*will then fail. However, because of the functional dependencies, there is no need to look for other managers of the project Turing. A compiler can use the functional dependencies to prevent unnecessary backtracking.* $\Box$

In Section 6, we show how the functional dependencies may be generalised to allow for functions and constructed terms in the language.

The definition of functionality implies that, if it exists, the correct answer for a functional goal must be ground.

**Example 3.3** *Consider the program*

$P(x)$.

*Declaratively,* $\forall y P(y)$ *is a logical consequence of the program. However, provided the underlying language has at least one ground term, say constant* $A$*, the goal* $\leftarrow P(y)$ *has correct answers* $y = x$ *and* $y = A$ *so that the goal is not functional.* □

To allow for non-ground answers, we need to consider the procedural semantics. Hence, an alternative form of determinacy based on computed rather than correct answers is often more useful. Note that functional computations, as defined in [19], are primarily concerned with the uniqueness of the computed answer. Determinacy based on the computed answer and thus the definitions in [19] will be considered later in Section 5.

# 4 Structural Determinacy

In contrast to the previous approach where the computation was regarded as a black box, in this and the following section, behaviour is characterised procedurally by SLDNF-trees. This section is concerned only with the structure of the tree while the next section is concerned with the variable bindings that label the branches of the tree.

Structural determinacy may be defined *locally*, that is in relation to a single node or just a small component of the tree, or *globally*, where the whole search tree is considered. First, local determinacy is examined.

**Definition** Given a program $\Pi$ with selection rule $\mathcal{S}$, then an atom $A$ is *head determinate in* $\Pi$ if at most one clause head in $\Pi$ unifies with $A$. A goal $G$ for $\Pi$ is *head determinate* (wrt $\mathcal{S}$) if each selectable positive literal in $G$ is head determinate in $\Pi$. A predicate $P/n$ in $\Pi$ is *head determinate* (wrt $\mathcal{S}$) if each goal of the form $\leftarrow\ P(t_1, \ldots, t_n)$ is head determinate.

**Example 4.1** *This example (which is a simplified form of a program in [38]) checks that the brackets in a list of brackets are balanced.*

$Balanced(l)\ \leftarrow\ Balanced(l, stack)$

$Balanced([\,], [\,])$
$Balanced([\text{"("}|ls], ss)\ \leftarrow\ Balanced(ls, [\text{"("}|ss])$
$Balanced([\text{"\{"}|ls], ss)\ \leftarrow\ Balanced(ls, [\text{"\{"}|ss])$
$Balanced([\text{")"}|ls], [\text{"("}|ss])\ \leftarrow\ Balanced(ls, ss)$
$Balanced([\text{"\}"}|ls], [\text{"\{"}|ss])\ \leftarrow\ Balanced(ls, ss).$

*Then provided Balanced(t) is selectable only when t is ground, the predicate Balanced/2 is head determinate.* □

The concept of *head determinacy* was first defined in [26] and more recently has been described as the "more basic form of determinacy" [10]. A concept closely related to head determinacy is ground determinacy; a predicate $P/n$ is *ground determinate* if every ground atom of the form $P(t_1, \ldots, t_n)$ unifies with at most one clause head [32][2]. A program is *ground determinate* if every

[2] In [32], ground determinacy is called local determinacy. (The name is changed here to avoid confusion with the concept of local structural determinacy.)

predicate in the program is ground determinate. For example, in Example 4.1, *Balanced*/2 and, hence the whole program, is ground determinate. It has been shown that for a program that is either ground determinate or subsumption equivalent to a program that is ground determinate, finite failure implies falsity [32][3]. In [26], head determinacy is extended to *primitive determinacy* where the unification criteria for a head determinate atom or goal is extended to allow for *primitives*. These are atoms which are known to be ground before they are selected. In [47], head determinacy is extended to allow for *test unifications* which are unifications, generated by solving equations in the body of a clause, that do not bind variables in the head. Here, we define a *test literal*, which is a literal in the body of a clause whose execution does not bind variables in the head. This generalises both the above concepts of a primitive atom and a test unification.

**Definition** Let $\Pi$ be a program with selection rule $\mathcal{S}$ and $C$ the clause

$A \leftarrow L_1 \wedge \cdots \wedge L_l$

Then $L_i$, $i \in \{1, \ldots, l\}$, is a *test literal of $C$ (wrt $\mathcal{S}$)* if, for each substitution $\theta$ such that $A\theta$ is selectable, $L_i\theta$ contains no variables in $A\theta$ and is selectable in $\leftarrow (L_1\theta \wedge \cdots \wedge L_l\theta)$. If $T$ is a conjunction of test literals for $C$, then $\leftarrow T$ is a *test goal for $C$ (wrt $\mathcal{S}$)*. A *test equation* for $C$ is a test literal for $C$ of the form $s = t$.

**Definition** Let $\Pi$ be a program with a selection rule $\mathcal{S}$, $A$ an atom, and $C_1, \ldots, C_k$ clauses in $\Pi$ whose heads unify with $A$ with substitutions $\theta_1, \ldots, \theta_k$, respectively. Suppose $\leftarrow T_1, \ldots, \leftarrow T_k$ are test goals for $C_1, \ldots, C_k$, respectively. Then $A$ is *primitive determinate* if at most one of the formulas $\forall T_1\theta_1, \ldots, \forall T_k\theta_k$ is a logical consequence of comp($\Pi$). The goal $G$ is *primitive determinate* (wrt $\mathcal{S}$) if each selectable positive literal in $G$ is primitive determinate in $\Pi$. A predicate $P/n$ is *primitive determinate* (wrt $\mathcal{S}$) if each goal of the form $\leftarrow P(t_1, \ldots, t_n)$ is primitive determinate.

The following example illustrates the difference between head and primitive determinacy.

**Example 4.2** *Consider the following program.*

$P(x) \leftarrow Q(x)$
$P(x) \leftarrow R(x, y) \wedge Q(y)$
$Q(1)$
$R(2, 1)$.

*Provided the atom $P(t)$ is selectable only when $t$ is ground, $Q(x)$ and $R(x, y)$ are test atoms in the first clauses and the predicate $P/1$ will be primitive determinate but not head determinate.* $\square$

In general, the property of primitive determinacy is undecidable [26]. The next proposition states some basic properties of head and primitive determinacy.

**Proposition 4.1** *Let $\Pi$ be a program and $\mathcal{S}$ a selection rule for $\Pi$. Then,*

1. *if a goal or predicate is head (resp., primitive) determinate wrt $\mathcal{S}$, it is head (resp., primitive) determinate wrt any stronger selection rule;*
2. *if goals $\leftarrow W$ and $\leftarrow W_1$ are head (resp., primitive) determinate wrt $\mathcal{S}$, the goal $\leftarrow W \wedge W_1$ is head (resp., primitive) determinate wrt $\mathcal{S}$.*

[3] Definite programs $\Pi_1$ and $\Pi_2$ are *subsumption equivalent* if every clause in $\Pi_1$ is subsumed by a clause in $\Pi_2$ and vice versa. A definite clauses $A_1 \leftarrow B_1$ is subsumed by a definite clause $A_2 \leftarrow B_2$, if $A_1 = A_2\theta$ and the set of atoms in the body $B_2\theta$ is a subset of set of atoms in $B_1$.

□

**Proof** Part 1 is obvious. For 2, note that by the first condition in the definition of a selection rule, any selectable atom in the goal $\leftarrow W \wedge W_1$ is also selectable in $\leftarrow W$ or $\leftarrow W_1$. □

Note that, unlike functionality, with a flexible selection rule a goal $\leftarrow W$ may be head determinate but an instance $\leftarrow W\theta$ may not be. The following example illustrates this.

**Example 4.3** *Consider the program*

$P(1, A)$
$P(1, B)$
$Q(1, A)$

*where $P(s, t)$ is selectable only when $s$ is ground. Then the goal*

$\leftarrow\ Q(x, y)\ \wedge\ P(x, y)$

*is head determinate, although, as the selection rule allows $P(1, y)$ to be selected before $Q(1, y)$, the goal*

$\leftarrow\ Q(1, y)\ \wedge\ P(1, y)$

*is not head determinate.* □

Costa refines head determinacy to *flat determinacy* [10]. Flat determinacy is similar to primitive determinacy except that test goals are restricted to be conjunctions of built-in atomic tests such as arithmetic comparisons, term comparisons, and built-in type tests.

**Example 4.4** *Consider the following program for partitioning a list (as used in quicksort).*

$Partition(\_, [\,], [\,], [\,])$
$Partition(i, [x|xs], [x|ys], zs)\ \leftarrow\ x \geq i\ \wedge\ Partition(i, xs, ys, zs)$
$Partition(i, [x|xs], ys, [x|zs])\ \leftarrow\ x < i\ \wedge\ Partition(i, xs, ys, zs).$

*Provided the atom $Partition(t_1, t_2, t_3, t_4)$ is selectable only when $t_1$ is ground and $t_2$ is a list which is either empty or whose first element is ground, then the predicate $Partition/4$ will be flat determinate but not head determinate.* □

The concept of head determinacy and its relationship with flat and primitive determinacy are further illustrated in the following example [10].

**Example 4.5** *Consider the following program.*

$P([\,], \_, \_, [\,])$
$P([x|xs], p, y, [x|ny])\ \leftarrow\ x > p\ \wedge\ P(xs, p, y, ny)$
$P([x|xs], p, y, [x|ny])\ \leftarrow\ x \leq p\ \wedge\ Member(x, y)\ \wedge\ P(xs, p, y, ny)$
$P([x|xs], p, y, ny)\ \leftarrow\ x \leq p\ \wedge\ \neg Member(x, y)\ \wedge\ P(xs, p, y, ny).$

*If the first argument of $P/4$ is a list of numbers, $t$; the second argument is a number, $p$; and the third is another list of numbers, $u$; then $P$ is true if the fourth argument corresponds to the list obtained from $t$ by deleting those elements which are either $\leq p$ or members of the list $u$. For any selection rule, the goal*

$\leftarrow\ P([\,], 5, [2], l)$

*is head determinate, and hence both flat and primitive determinate. In the goal*

$\leftarrow\ P([7], 5, [2], l)$

*the atom* $P([7], 5, [2], l)$ *unifies with the heads of the last three clauses and hence is not head determinate. The call* $7 \leq 5$*, however, will fail so that only the second clause will succeed. Thus this goal is both flat and primitive determinate. In the goal*

$\leftarrow\ P([5], 7, [2], l)$

*the atom* $P([5], 7, [2], l)$ *unifies with the heads of the last three clauses. Moreover, the test* $5 \leq 7$ *will succeed in both the third and fourth clauses so that the goal is not flat deterministic. However, the test* $Member(5, [2])$ *in the third clause will fail so that only the fourth clause will succeed. Thus this goal is primitive determinate.*

*This illustrates that, provided atoms of the form* $P(t_1, t_2, t_3, t_4)$ *are selectable only if* $t_1$ *and* $t_2$ *are ground, the predicate* $P/4$ *is primitive but not flat or head determinate.* □

The test literals $Member(x, ls)$ and $\neg Member(x, ls)$ occur in different clauses and effectively define a two way switch. This use of an atom and its negation as test literals in different clauses of a predicate is relatively common. A number of languages such as Gödel and NU-Prolog thus provide a special *If-Then-Else* construct to make such a test explicit (see Section 7).

More pragmatic approaches to structural determinacy define determinacy with respect to a clause [50, 52]. This form of determinacy is really a refinement of primitive determinacy that assumes a fixed search strategy. In the following definition, it is assumed that there is a ordering $<$ on the clauses in a definition and that the search strategy respects this order. Thus, if a node has child nodes $n$ and $n'$ with clauses $C$ and $C'$, respectively, such that $C < C'$ then the node $n$ is visited before $n'$.

**Definition** Let $\Pi$ be a program with selection rule $\mathcal{S}$ and $C$ a clause in $\Pi$ with test goal $\leftarrow T$. Suppose that, for each selectable atom $A$ that unifies with the head of $C$ with substitution $\theta$ and $\leftarrow T\theta$ succeeds, whenever $A$ also unifies with the head of another clause $C' > C$ with test goal $\leftarrow T'$ with substitution $\theta'$, $\leftarrow T'\theta'$ fails. Then $C$ is said to be *clause determinate*[4].

Thus the maximal clause in the ordered sequence of clauses defining a predicate is always determinate. In [50], a test in a clause is defined to be a literal in the body of the clause whose execution will not cause side effects, will not create choice points, only binds variables local to the clause, and does not create structures on the heap. Although, this definition depends on the compiled form of the test rather than the test itself, it appears that the intended meaning of a test is close, in spirit, to the tests allowed in primitive determinacy. In [52], (where a determinate clause is called a *cut clause*) a test goal is restricted to one whose body is a conjunction of built-in atomic tests and a predicate is said to be *deterministic* if every clause in its definition is a cut clause. The idea of a determinate clause is illustrated in the following example.

**Example 4.6** *Consider a program of the form:*

[4] This definition is a modified form of that given in [50] and it does not take into account cuts or side-effect predicates.

$P(A, x) \leftarrow \ldots$
$P(B, x) \leftarrow \neg(x = F(\_)) \wedge \ldots$
$P(B, F(3)) \leftarrow \ldots$
$P(B, F(4)) \leftarrow \ldots.$

*Then, assuming atoms of the form $P(t_1, t_2)$ are selectable when both arguments are non-variables, the first, second and fourth clauses are determinate, whereas the third clause is not. The first clause is determinate since any selectable atom that unifies with $P(A, x)$ cannot unify with any following clauses. The second clause is determinate since the test $\neg(x = F(\_))$ implies that none of the following clauses will unify with the atom. The fourth clause is determinate because it is the last clause.* □

Information concerning the determinacy of a clause can be used to remove unnecessary choice points as early as possible during execution. By exploiting clause determinacy in a special compilation scheme, speedups of between 30% and 300% were reported for a number of test programs [50].

Global forms of determinacy for which the SLDNF-tree for a program and goal must approximate to a single branch require the corresponding local form to hold at every node of the SLDNF-tree.

**Definition** Given a program $\Pi$ with selection rule $\mathcal{S}$ and a goal $G$ for $\Pi$. Then an SLDNF-derivation for $\Pi \cup \{G\}$ is *head* (resp., *primitive*) *determinate* wrt a selection rule $\mathcal{S}$, if the derivation uses $\mathcal{S}$ and each selected atom in the derivation is head (resp., primitive) determinate wrt $\mathcal{S}$. The goal $G$ is *derivation head* (resp., *primitive*) *determinate* (wrt $\mathcal{S}$) if every SLDNF-derivation of $\Pi \cup \{G\}$ using $\mathcal{S}$ is head (resp., primitive) determinate wrt $\mathcal{S}$.

Thus, if $G$ is derivation head determinate wrt $\mathcal{S}$, then every goal in an SLDNF-tree for $\Pi \cup \{G\}$ using $\mathcal{S}$ will be head determinate wrt $\mathcal{S}$ and every SLDNF-tree for $\Pi \cup \{G\}$ using $\mathcal{S}$ will have a single branch.

**Example 4.7** *Consider again the program for Partition/4 in Example 4.4. If $t_1$ and $t_2$ are ground terms, then the goal $\leftarrow Partition(t_1, t_2, t_3, t_4)$ is derivation primitive determinate.* □

**Proposition 4.2** *Let $\Pi$ be a program and $\mathcal{S}$ a selection rule for $\Pi$. Then, if a goal or predicate is head (resp., primitive) determinate wrt $\mathcal{S}$, it is head (resp., primitive) determinate wrt any stronger selection rule.* □

**Proof** Obvious. □

Note that unlike head determinacy, goals $\leftarrow W$ and $\leftarrow W_1$ may be derivation head determinate but the goal $\leftarrow W \wedge W_1$ may not be.

**Example 4.8** *Consider the program*

$P(1, A)$
$P(1, B)$
$Q(z, A) \leftarrow S(z)$
$R(1)$
$S(1)$

*with selection rule such that $P(s, t)$ is selectable only if the term $s$ is ground. Then the goals*

$\leftarrow\ Q(x, y)\ \wedge\ P(x, y)$
$\leftarrow\ R(x)$

*are derivation head determinate, while the goal*

$\leftarrow\ R(x)\ \wedge\ Q(x, y)\ \wedge\ P(x, y)$

*is not derivation head determinate.* □

As computed answers may not be ground, a goal which is derivation primitive determinate is not necessarily functional. Moreover, a goal may be functional and head determinate but not derivation head (or primitive) determinate.

**Example 4.9** *Consider the program*

$P(x, y)\ \leftarrow\ Q(x, y)\ \wedge\ R(x, y)$
$Q(A, B)$
$Q(A, A)$
$R(x, x).$

*The goal* $\leftarrow\ R(u, v)$ *is trivially derivation head determinate, but not functional. Furthermore, assuming a left to right selection rule, the goal* $\leftarrow P(u, v)$ *is head determinate and functional with the correct answer* $u = A, v = A$ *but not derivation head determinate.* □

An alternative notion for global determinacy, presented in [43], defines a deterministic goal to be such that there is at most one success branch at each node of the SLDNF-tree. Thus there can still be any number of failing or infinite branches in the tree.

**Example 4.10** *Thus, according to [43], for the program of Example 4.9, the goal*

$\leftarrow\ P(u, v)$

*is determinate. However, it is not derivation primitive determinate.* □

The development of a good search strategy that minimises backtracking requires knowledge of the structural determinacy for the program and goal. If a goal in a derivation is known to be head determinate then no choice point need be created at the corresponding node in the search tree. Moreover, if it is known that a goal is derivation head determinate, then no choice points in its derivation need be created. With primitive determinacy, choice points may still be made and only after evaluating the test goal for a clause would they be removed. A search procedure that detects and then removes many choice points where the unexplored branches do not lead to success is called *intelligent* [6, 12] or *selective backtracking* [40]. Detection of unnecessary choice points is normally performed at run-time during the forward execution [8]. If the detection is based on compile-time analysis, the backtracking is called *semi-intelligent* [9]. A hybrid scheme that relies on both static and run-time analysis that was developed for exploiting parallelisation can also be used to underpin intelligent backtracking [51]. A more challenging analysis problem is that of reducing the overheads of dynamic backtracking [23]. Dynamic backtracking, unlike intelligent and semi-intelligent backtracking, can move backtrack points deep into the search space so as to avoid redoing useful work.

# 5 Substitutional Determinacy

In Section 3, the declarative form of determinacy called functionality was defined. One limitation of this definition was that, as a functional goal can have at most one correct answer, this answer had to be ground. To consider non-ground answers, the definition of determinacy needs to be based on the actual answers that are computed. To distinguish functionality from determinacy that is based on the computed answer, we call the computational form *solution determinacy*.

**Definition** Given a program $\Pi$ with selection rule $\mathcal{S}$, then a goal $G$ is *solution determinate* wrt $\mathcal{S}$ if there is (up to equivalence) at most one computed answer using $\mathcal{S}$ for $\Pi \cup \{G\}$. A predicate $P/n$ is *solution determinate* wrt $\mathcal{S}$ if every goal of the form $\leftarrow P(t_1, \ldots, t_n)$, where $P(t_1, \ldots, t_n)$ is selectable wrt $\mathcal{S}$, is solution determinate.

The next proposition is similar to Proposition 3.1 and states some basic properties of solution determinacy.

**Proposition 5.1** *Let $\Pi$ be a program and $\mathcal{S}$ a selection rule for $\Pi$. Then,*

1. *if a goal or predicate is solution determinate wrt $\mathcal{S}$, it is solution determinate wrt any stronger selection rule;*
2. *if a goal $\leftarrow W$ does not flounder for any SLDNF-derivation using $\mathcal{S}$ and is solution determinate wrt $\mathcal{S}$, the goal $\leftarrow W\theta$ is solution determinate wrt $\mathcal{S}$ for any substitution $\theta$;*
3. *if the goals $\leftarrow W$ and $\leftarrow W_1$ do not flounder for any SLDNF-derivation using $\mathcal{S}$ and are solution determinate wrt $\mathcal{S}$, the goal $\leftarrow W \wedge W_1$ is solution determinate wrt $\mathcal{S}$.*

$\square$

**Proof** 1 is obvious. For 2, suppose $\Pi \cup \{\leftarrow W\}$ has computed answer $\sigma$. Let $D$ be a derivation for $\Pi \cup \{\leftarrow W\theta\}$ with computed answer $\phi$. If $E$ is the set of equations $\{x = t : x/t \in \theta\}$, we can obtain a derivation $D'$ for $\leftarrow W \wedge E$ from $D$ by adding a partial derivation[5] $D_E$ which solves the equations in $E$ and ends with a variant of the goal $\leftarrow W\theta$ to the beginning of $D$. Then $D'$ has the same computed answer $\phi$. By [30, Theorem 9.2], a computed answer does not depend on the selected literal at each derivation step. Thus, as no derivation for $\leftarrow W$ flounders, we can obtain a derivation $D''$ from $D'$ with computed answer equivalent to $\phi$ by first solving $\leftarrow W$, selecting instances of equations in $E$ only when there are no other literals in the current goal. Thus, as $\sigma$ is the only computed answer for $\leftarrow W$, $D''$ has computed answer $mgu(E\sigma)$. As $D$ was an arbitrary derivation for $\leftarrow W\theta$, all computed answers for $\Pi \cup \{\leftarrow W\theta\}$ are equivalent to $mgu(E\sigma)$ and $\leftarrow W\theta$ is solution determinate.

For 3, suppose $\Pi \cup \{\leftarrow W\}$ has computed answer $\sigma$. Let $D$ be a derivation for $\Pi \cup \{\leftarrow W \wedge W_1\}$ with computed answer $\phi$. By [30, Theorem 9.2], a computed answer does not depend on the selected literal at each derivation step. Thus, as no derivation for $\Pi \cup \{\leftarrow W\}$ flounders, we can assume that $D$ consists of a partial derivation ending with a variant of the goal $\leftarrow W_1\sigma$ followed by a derivation for this variant. By Part 2, as $\leftarrow W_1$ is solution determinate, $\leftarrow W_1\sigma$ is also solution determinate and there is a unique computed answer $\psi$ for $\Pi \cup \{\leftarrow W_1\sigma\}$. Hence $D$ has computed answer $\sigma\psi$. As $D$ was an arbitrary derivation for $\leftarrow W \wedge W_1$ (and the order of the subgoals $W, W_1$ is not significant in the above part of the proof), all computed answers for $\Pi \cup \{\leftarrow W \wedge W_1\}$ are equivalent to $\sigma\psi$ and $\leftarrow W \wedge W_1$ is solution determinate. $\square$

[5] A *partial derivation* may end in a non-empty goal that does not fail or flounder.

Thus, any goal whose literals are either negative or atoms whose predicates are solution determinate will also be solution determinate. Note that the floundering condition in Parts 2 and 3 cannot be dropped or weakened.

**Example 5.1** *Consider the program*

$P(1, A)$
$P(1, B)$
$Q(1, A)$
$Q(z, u)$
$R(1)$
$S(1)$

*with selection rule such that $P(s, t)$ is selectable only if $s$ is ground. Then the goals*

$\leftarrow\ Q(x, y)\ \wedge\ P(x, y)$
$\leftarrow\ R(x)$

*are solution determinate as the branch using the fact $Q(z, u)$ will flounder while the goals*

$\leftarrow\ Q(1, y)\ \wedge\ P(1, y)$
$\leftarrow\ R(x)\ \wedge\ Q(x, y)\ \wedge\ P(x, y)$

*are not solution determinate.* □

As SLDNF-resolution is sound, any predicate that is functional (in the sense of Section 3) is also solution determinate. However, Example 3.3 shows that the converse does not hold.

If a goal is derivation head (or primitive) determinate, there is at most one branch in any SLDNF-tree for the goal and hence, at most one computed answer. Thus the goal is also solution determinate. However, the following example shows that the converse does not hold.

**Example 5.2** *Consider the following program.*

$P(x, y)\ \leftarrow\ Q(x)$
$P(x, y)\ \leftarrow\ R(x)$
$R(A)$
$Q(A)$.

*With the trivial selection rule, the goal $\leftarrow P(u, v)$ and predicate $P/2$ are solution determinate but not head or primitive determinate. Note also, as the goal $\leftarrow P(u, v)$ has more than one correct answer, $P/2$ is not functional.* □

A definition similar to that of solution determinacy is given in [19] and it is observed that, in general, this form of determinacy is undecidable. In [26], it was been noted that if SLD-resolution is considered as a goal rewriting process, then a goal will be solution determinate if and only if it is Church-Rosser with respect to the substitutions returned for the variables of the goal.

In the logic programming language Mercury [47], a mode of a predicate is categorised according to the number of computed answers that are obtained for an atomic query with that predicate and which satisfy the given mode. In particular, a mode for a predicate $P/n$ is said to be *deterministic* if all queries of the form $\leftarrow P(t_1, \ldots, t_n)$ such that $P(t_1, \ldots, t_n)$ satisfies the mode, have exactly one computed answer. There are similar definitions for *semideterministic*, *multideterministic*, and *nondeterministic* modes. Both the modes and their determinacy category must be declared by the user. This information together with other declared information is used to produce very efficient compiled code.

In a parallel computation, a binding may be made and communicated to processes running on other processors. The computation with this binding might then subsequently fail so that work done by the other processes would have to be undone. This form of distributed backtracking is difficult to implement and it is desirable to avoid it.

**Definition** Let $\Pi$ be a program with selection rule $\mathcal{S}$. The goal $G$ is *binding determinate* if $\Pi \cup \{G\}$ has computed answer $\theta$ and, for any SLDNF-tree for $\Pi \cup \{G\}$ using $\mathcal{S}$ and any composition $\phi$ of substitutions on a branch in the tree, there exists a substitution $\psi$ such that the restriction of $\phi\psi$ to the set of variables in $G$ is $\theta$.

The essential idea in binding determinacy is that the bindings for the variables in the goal never need to be retracted.

**Example 5.3** *Consider the following program.*

$P(F(x, y)) \leftarrow Q(x, y)$
$P(F(x, y)) \leftarrow Q(y, x) \wedge Q(x, y)$
$Q(A, B)$

*and goals*

$\leftarrow P(F(A, y))$
$\leftarrow P(y)$

*Then the first but not the second goal is binding determinate.* □

**Proposition 5.2** *Let $\Pi$ be a program and $\mathcal{S}$ a selection rule for $\Pi$. Then, if a goal or predicate is binding determinate wrt $\mathcal{S}$, it is binding determinate wrt any stronger selection rule.* □

**Proof** Obvious. □

Note that unlike solution determinacy, even if no derivations flounder, goals $\leftarrow W$ and $\leftarrow W_1$ may be binding determinate but the goal $\leftarrow W \wedge W_1$ may not be.

**Example 5.4** *Consider again the program in Example 4.8. Then the goals*

$\leftarrow Q(x, y) \wedge P(x, y)$
$\leftarrow R(x)$

*are binding determinate while the goal*

$\leftarrow R(x) \wedge Q(x, y) \wedge P(x, y)$

*is not binding determinate.* □

Unlike solution determinacy, a goal may be binding determinate but an instance may not be.

**Example 5.5** *Consider again the program in Example 4.3. Then the goal*

$\leftarrow Q(x, y) \wedge P(x, y)$

*is also binding determinate, although the goal*

$\leftarrow\ Q(1, y)\ \wedge\ P(1, y)$

*is not.* □

Solution determinacy for a goal is not as dependent as binding determinacy on the computation rule. The following example illustrates this.

**Example 5.6** *This program finds the minimum of two numbers.*

$$Min(x, y, m)\ \leftarrow\ x \leq y\ \wedge\ m = x$$
$$Min(x, y, m)\ \leftarrow\ y \leq x\ \wedge\ m = y.$$

*Then, assuming a right to left selection rule, the goal* $\leftarrow Min(2, 1, n)$ *is solution determinate (and functional) but not binding determinate. However, with a selection rule that selects from left to right, the goal is binding determinate. Note that as* $x \leq y$ *and* $y \leq x$ *both succeed when* $x$ *and* $y$ *are equal, Min/3 is not primitive determinate.* □

If a goal is derivation head (or primitive) determinate, there is at most one branch in any SLDNF-tree for the goal and thus the goal is also binding and solution determinate. However, the converse does not hold.

**Example 5.7** *Consider again the program in Example 5.2. The the goal*

$\leftarrow\ P(u, v)$

*is binding determinate but not derivation head or primitive determinate.* □

Predicates that are binding deterministic can be executed using dependent and-parallelism without the need for distributed backtracking. This is the rationale behind parallel NU-Prolog [38]. A variation of binding determinacy is also used in Reform Prolog [4], an implementation of Prolog that uses recursion parallelism. Recursion parallelism is a special form of dependent and-parallelism restricted to just the linearly, structurally recursive predicates. The computation for a goal with recursion depth $n$ is translated into $n$ goal-head unifications proceeded by $n$ body computations. Reform compilation [35] is thus not unlike loop parallelisation. The recursive predicates for parallelisation are explicitly requested by the programmer while the compiler checks that backtracking is not distributed between processors. To prevent this, the compiler must ensure that all bindings for variables which are possibly shared across parallel recursive iterations can be made unconditionally.

Determinacy analysis also underpins the loop parallelisation work in [44] where a series of (control-flow graph) transforms are proposed to expose loops (recursive predicates) in which each iteration of the loop can be executed in parallel. Two forms of loop parallelisation are used. The first parallelisation, unlike Reform, does not require shared variables to be binding determinate. Although very general, the implementation is complicated because of the way bindings have to be handled. Therefore a top-down mode analysis is used to recognise (head) determinate predicates and thus detect cases of binding determinacy. This permits a second, more efficient, Reform type of parallelisation to be used wherever possible.

One possible application of binding determinacy that has not previously been considered is in dynamic dependent and-parallelism (DDAS) [45, 46]. This can execute full Prolog programs using both independent and dependent and-parallelism. The cost of supporting the complexity of independent and-parallelism, however, is only incurred when the parallelism is actually exploited.

The fixed left to right selection rule of Prolog forces a fixed order for the unifications occurring in a derivation. In DDAS, this unification order is maintained. To achieve this, variables which are potentially shared between atoms in the body of a clause and may be non-ground when executed are labelled as dependent variables. The Prolog unification order is preserved by only allowing the process executing the leftmost atom that shares a dependent variable (the producer) to bind that variable. When a process executing another atom (a consumer) attempts to bind the shared variable to a non-variable term, the process will be suspended until either the variable becomes bound (by the producer process), or the atom becomes the leftmost atom sharing the variable. The problem here is that, if the producer process unbinds a dependent variable, the consumer processors may need to re-do their computation. Currently a simple run-time mechanism is used in DDAS to keep track of the consumption of bindings to help reduce this wasted work. If, however, it was known that a subgoal consisting of the producer and some of the consumers was solution determinate but not binding determinate (with respect to the dependent variable), then the execution of the remaining consumers could be delayed until this subgoal had successfully terminated. Although this would reduce the amount of parallelism, it would also reduce wasted work and hence improve the efficiency of DDAS. To illustrate this, consider the following example.

**Example 5.8** *The predicate Connected/2 finds the connected nodes in a directed (possibly cyclic) graph. This is defined using a subsidiary predicate Connect/3 which, to prevent looping, keeps a record of all the nodes that have been visited and checks that no node is visited more than once.*

$Connected(x, y) \ \leftarrow \ Connect(x, y, [x])$

$Connect(a, a, \_)$
$Connect(a, b, v) \ \leftarrow \ Edge(a, n) \ \wedge \neg Member(n, v) \ \wedge \ Connect(n, b, [n|v])$
$Edge(1, 1)$
$Edge(1, 2)$.

*The second clause for Connect/3 has three literals in its body which may be executed in parallel. By considering the variable $n$ in this clause, it can be seen that $Edge(a, n)$ is the producer and $\neg Member(n, [v])$ and $Connect(n, b, [n|v])$ are consumers. With the goal*

$\leftarrow \ Connected(1, 2)$

*the second clause for Connect/3 will be chosen with $a$ bound to 1, $v$ to [1], and $b$ to 2. Assuming the clauses are tried in textual order, $n$ will be bound to 1 by the process for $Edge(1, n)$, the producer for $n$. This binding, will enable the consumer processes for $n$ to start up. However, the process for $\neg Member(1, [1])$ will fail and the process for $Edge(1, n)$ will thus search for the alternative binding $n/2$. The remaining consumer process for $Connect(n, 2, [n|1])$ will now have to redo its computation with the new value 2 for $n$. It is thus desirable that the execution of the consumer $Connect(n, 2, [n|1])$ should be delayed until $\leftarrow \ Edge(1, n) \ \wedge \neg Member(n, [1])$ has succeeded. This will prevent unnecessary work.* □

# 6 Static Analysis for Determinacy

In this section, we consider a number of published methods for inferring the different forms of determinacy. The analysis for any form of determinacy may be done entirely at compile time (*static analysis*), entirely at run time (*dynamic analysis*), or at both compile time and run time (*hybrid analysis*). Moreover the analysis may be *passive*, where the selection rule is fixed by the program and its programming language or *proactive*, determining restrictions for any underlying selection rule so as to ensure certain goals are determinate. In this section, we outline and compare some of the different static analysis techniques that have been described.

This section is subdivided to correspond with each of the main procedural forms of determinacy; structural, solution, and binding determinacy. We do not consider the declarative concept of functionality by itself as the analysis methods are all based on the procedural view of logic programming. Moreover, solution determinacy implies functionality, so that methods that derive the solution determinacy of a predicate can also infer its functionality. There are many analysis techniques for inferring modes from known selection rules. These techniques are not considered and in this section it is assumed that the modes of use of the predicates have already been ascertained by such techniques.

## Structural Determinacy

We describe a method of analysis for head and primitive determinacy which is both static and passive and based on the method defined in [26].

Inferring head determinacy for a predicate in a program is a simple matter once the modes of use of the predicate are known. Terms in the heads of clauses defining the predicate that are not at input positions are first replaced by unique variables and then pairs of these modified clause heads (standardised apart so that they have no variables in common) are checked that they do not unify. Note that, to obtain accurate results, this procedure should use, not only the modes of a predicate's arguments, but also the modes of argument positions in constructed terms.

**Example 6.1** *Consider the following program.*

$Split([\,],[\,],[\,])$
$Split([h],[h],[\,])$
$Split([a,b|r],[a|s],[b|t]) \leftarrow Split(r,s,t).$

*If it is only known that the first argument is non-variable and, if we replace all non-input positions by unique variables, we have the following atoms derived from the clause heads.*

$Split([\,],x1,y1)$
$Split([h|w2],x2,y2)$
$Split([h|w3],x3,y3).$

*Thus, with this selection rule, the predicate $Split/3$ will not be head determinate. However, if the selection rule ensured that whenever the list in the first argument was non-empty, the tail of the list would be a non-variable, then the atoms derived from the clause heads would be*

$Split([\,],x1,y1)$
$Split([h],x2,y2)$
$Split([a,b|w],x3,y3)$

*and $Split/3$ would be head determinate.* □

The next lemma shows that the procedure for head determinacy is safe.

**Lemma 6.1** *Let $\Pi$ be a program with selection rule $\mathcal{S}$, $A_1 \leftarrow B_1,\ldots,A_k \leftarrow B_k$ clauses defining a predicate $P$, and $A'_1,\ldots,A'_k$ atoms obtained from $A_1,\ldots,A_k$, respectively, using the technique described above. Then, if $A'_i$ and $A'_j$, for all $i \neq j \in \{1,\ldots,k\}$, are not unifiable, $P$ is head determinate wrt $\mathcal{S}$.*

**Proof** If a selectable atom $A$ unifies with $A_i$, $i \in \{1, \ldots, k\}$, then it is an instance of $A'_i$. Thus, if for $i, j \in \{1, \ldots, k\}$, $A'_i$ and $A'_j$ are not unifiable, $A$ can unify with at most one of $A_i, A_j$. $\square$

This algorithm can be extended to allow for some primitive determinacy. First, all pairs of heads of clauses in the definition of a predicate for which the above unifiability test succeeds (and hence the determinacy test fails) are found. If there are no such pairs, then the predicate is primitive determinate. Otherwise, suppose that for each pair $\{C_1, C_2\}$ of clauses for which the unifiability test succeeds with a substitution, say $\theta$, there are test goals $\leftarrow T_1$ and $\leftarrow T_2$ for $C_1$ and $C_2$, respectively, such that $\forall\neg(T_1\theta \;\wedge\; T_2\theta)$ is a logical consequence of the program, then the predicate is primitive determinate. Apart from the obvious built-in tests such as arithmetic comparisons, an atom and its negation are often used as tests for primitive determinacy.

**Example 6.2** *Consider the following program.*

$Sort([\,], [\,])$
$Sort([x|xs], ys) \;\leftarrow\; Sort(xs, zs) \;\wedge\; Insert(x, zs, ys)$

$Insert(x, [\,], [x])$
$Insert(x, [y|ys], [y|zs]) \;\leftarrow\; x > y \;\wedge\; Insert(x, ys, zs)$
$Insert(x, [y|ys], [x, y|zs]) \;\leftarrow\; x \leq y.$

*If it is known that the first argument to Sort/2 is always ground and that the selection of atoms is left-to-right, it can be inferred that the first two arguments of Insert will be ground. Thus the modified clause heads for the head determinacy test are:*

$Sort([\,], y1)$
$Sort([x|xs], y2)$
$Insert(x, [\,], z1)$
$Insert(x, [y|ys], z2)$
$Insert(x, [y|ys], z3).$

*Then all but the last two heads will not unify. However, as $x$ and $y$ will be ground, $x > y$ and $x \leq y$ are test atoms in these clauses and $\forall\neg(x > y \;\wedge\; x \leq y)$ is a logical consequence of (the completion of) the program. (We assume the axioms of arithmetic are included in the completed program.) Thus, in this case, both Sort/2 and Insert/3 will be found to be primitive determinate.* $\square$

The next lemma shows that the above procedure for primitive determinacy is safe.

**Lemma 6.2** *Let $\Pi$ be a program with selection rule $\mathcal{S}$, $A_1 \leftarrow T_1 \wedge B_1, \ldots, A_k \leftarrow T_k \wedge B_k$ be clauses defining a predicate $P$ in $\Pi$, and $A'_1, \ldots, A'_m$ the modified heads as described in the procedure for head determinacy. Suppose, for each $i \neq j \in \{1, \ldots, k\}$, whenever $A'_i$ and $A'_j$ are unifiable with substitution $\theta_{i,j}$, $\forall\neg(T_i\theta_{ij} \wedge T_j\theta_{ij})$ is a logical consequence of* comp($\Pi$). *Then $P$ is primitive determinate.*

**Proof** If $A'_i$ and $A'_j$ are not unifiable, then, as in the proof of Proposition 6.1, a selectable atom can unify with at most one of $A_i, A_j$. If $A'_i$ and $A'_j$ are unifiable and a selected atom $A$ unifies with both $A_i$ and $A_j$, then it is an instance of $A'_i\theta_{ij}$. Thus $A$ unifies with $A_i, A_j$ with substitutions of the form $\theta_{ij}\phi_i, \theta_{ij}\phi_j$, respectively. By the hypothesis, at most one of the formulas $\forall T_i\theta_{ij}\phi_i, \forall T_j\theta_{ij}\phi_j$ is a logical consequence of comp($\Pi$). Then, as our choice of selected atom $A$ and the heads of clauses $A_i, A_j$ that unify with $A$ were arbitrary, $P$ is primitive determinate. $\square$

Clearly, knowledge of the number of success branches for an SLDNF-tree for a program and goal subsumes knowledge of the derivation determinacy of the goal. Analysis to find an approximation to this number has been described in [41]. Here the analyser has an abstract domain $\{0, \mathcal{L}, 1, 1', 2, 2'\}$. The number 0 indicates zero, 1 one, and 2 at least 2 success branches. $\mathcal{L}$ indicates that it loops, 1' one solution and then it loops, and 2' two solutions and then it loops. The method which is primarily designed to find where the cut can be safely removed from Prolog programs is intended for use with a partial evaluator where the cut prevents unfolding.

## Solution Determinacy

We describe a method, based on the approach in [20], for detecting solution determinacy that is both static and passive. To explain this, we will require the concept called *functional dependency*. Functional dependencies are well-known in relational database theory [49] and assert constraints on the possible values for a relation. In most adaptations of this concept for logic programming, the definition requires atoms to be of the form $P(x_1, \ldots, x_n)$ where $x_1, \ldots, x_n$ are distinct variables [34]. Below we give a more general definition so that it allows for constructed terms and is applicable for any control modes that may take these into account.

**Definition** Let $\Pi$ be a program, $W$ a conjunction of literals with free variables $X$, and $U$ and $V$ subsets of $X$. Then $U \rightarrow V$ *is a functional dependency for* $W$ and $W$ *satisfies* $U \rightarrow V$ if, for any substitution $\theta$ for which $U\theta$ is ground, the restrictions of the computed answers for $\Pi \cup \{\leftarrow W\theta\}$ to the set $V$ are equivalent. If $D$ is a set of functional dependencies that includes all (resp., just) functional dependencies for $W$, then $D$ is *complete* (resp., *sound*) for $W$.

**Example 6.3** *Consider the following program.*

$Q(1, 0, 2, 0)$
$Q(1, 0, 3, 1)$
$Q(2, 1, 2, 0)$
$Q(2, 1, 4, 1)$
$R(1, 2)$
$R(0, 2)$.

*Then, from the definitions of* $Q/4$ *and* $R/2$*, it can be seen that the functional dependencies for* $Q(x, y, u, v)$ *include* $\{x\} \rightarrow \{y\}$, $\{y\} \rightarrow \{x\}$ *and* $\{u\} \rightarrow \{v\}$ *and a functional dependency for* $R(y, u)$ *is* $\{y\} \rightarrow \{u\}$. □

Given a set of functional dependencies for a set of variables, other dependencies will also hold.

**Example 6.4** *Given the set of functional dependencies*

$\{\{x\} \rightarrow \{y\}, \{y\} \rightarrow \{x\}, \{u\} \rightarrow \{v\}, \{y\} \rightarrow \{u\}\}$

*it is clear that the dependencies such as* $\{x\} \rightarrow \{u\}$ *and* $\{x\} \rightarrow \{x, u, v\}$ *also hold.* □

**Definition** Let $D$ be a set of functional dependencies and $U \rightarrow V$ a functional dependency, then $D$ *logically implies* $U \rightarrow V$ if each conjunction of literals that satisfies all the functional dependencies in $D$ also satisfies $U \rightarrow V$. The set $D^+$ of all functional dependencies logically implied by $D$ is called the *closure* of $D$.

If $D$ is a set of functional dependencies, the closure $D^+$ can be inferred using the following rules called Armstrong's axioms [49]. Let $U_1, U_2, U_3$ be subsets of $X$.

*Reflexivity*. If $U_1 \supseteq U_2$, then $U_1 \to U_2 \in D^+$.
*Transitivity*. If $U_1 \to U_2, U_2 \to U_3 \in D^+$, then $U_1 \to U_3 \in D^+$.
*Augmentativity*. If $U_1 \to U_2 \in D^+$, then $U1 \cup U_3 \to U_2 \cup U_3 \in D^+$.

Armstrong's axioms are sound and complete with respect to the logically implies relation defined above [49].

**Example 6.5** *Using the dependencies in Example 6.4, the dependency* $\{x\} \to \{u\}$ *follows from the transitivity axiom. The* $\{x\} \to \{x, u, v\}$ *dependency can be deduced from the dependencies in Example 6.4:*

1. $\{x\} \to \{u\}$ *(transitivity applied to given dependencies)*
2. $\{x\} \to \{x, u\}$ *(augmentativity of 1 by* $\{x\}$*)*
3. $\{x\} \to \{v\}$ *(transitivity applied to 1 and a given dependency)*
4. $\{x, u\} \to \{x, u, v\}$ *(augmentativity of 3 by* $\{x, u\}$*)*
5. $\{x\} \to \{x, u, v\}$ *(transitivity applied to 2 and 4)*

□

Given a set of functional dependencies for the atoms, functional dependencies for the body of a clause or goal can be inferred using Armstrong's axioms. In particular, if $D_1^+$ and $D_2^+$ are complete sets of functional dependencies for atoms $A_1$ and $A_2$ respectively, then $(D_1 \cup D_2)^+$ is a complete set of functional dependencies for $A_1 \wedge A_2$. Negative literals can be ignored, since they must be ground before they are selected.

**Example 6.6** *Consider again the program in Example 6.3. Then the complete set of functional dependencies for* $Q(x, y, u, v)$ *and* $R(y, u)$ *are, respectively,*

$\{\{x\} \to \{y\}, \{y\} \to \{x\}, \{u\} \to \{v\}\}^+$ *and*
$\{\{y\} \to \{u\}\}^+$.

*Thus the complete set of functional dependencies for* $Q(x, y, u, v) \wedge R(y, u)$ *is:*

$\{\{x\} \to \{y\}, \{y\} \to \{x\}, \{u\} \to \{v\}, \{y\} \to \{u\}\}^+$.

□

Functional dependencies have been used in work on deductive databases to optimise analysis techniques [31]. For such databases, although bottom-up is the preferred evaluation strategy, in the presence of non-ground facts this method may not be as efficient as top-down evaluation. The basic problem is that, to factor out any duplicates, the new facts that are computed at each iteration of the fixpoint calculation have to be compared with the previously generated facts. For non-ground facts, this requires a costly subsumption check unless the program has the special *subsumption-free* property [31]. Establishing subsumption-freedom requires functional dependencies to be derived over the relations defined by the program.

Interestingly, the notion of a functional dependency for a literal is analogous to the idea of functional constraints that have recently been applied in constraint satisfaction problems [13]. In a binary constraint network (a network in which constraints only relate pairs of variables, say, $x$

and $y$), a constraint is said to be functional if and only if $x \to y$ or $y \to x$. Functional constraints have been used to put better bounds on search complexity and, provided certain conditions hold, guarantee the existance of solutions [13].

The functional dependencies of a formula provide part of the criteria for a predicate to be solution determinate. To explain the other part, the concept of solution determinacy needs to be extended to clauses.

**Definition** Let $\Pi$ be program with selection rule $\mathcal{S}$. A clause $A \leftarrow B$ is *solution determinate* if, for every substitution $\theta$ such that an atom $A\theta$ is selectable in some goal for $\Pi$, the restrictions of the computed answers for $\leftarrow B\theta$ to the variables in $A\theta$ are all equivalent.

In [20], it is assumed that the functional dependencies of the atoms in the program are already known. However, as the definition of solution determinacy depends on a selection rule, we require the dependencies to be consistent with the given selection rule.

**Definition** Let $\Pi$ be a program with selection rule $\mathcal{S}$. A functional dependency $U \to V$ for an atom $A$ is *consistent with* $\mathcal{S}$ if, whenever $A\theta$ is selectable, $U\theta$ is ground.

A criteria for a clause to be solution determinate is as follows [20, Proposition 3.5].

**Proposition 6.1** *Let $A \leftarrow B$ be a clause in a program $\Pi$ with selection rule $\mathcal{S}$, $D$ a set of functional dependencies for $B$ that are consistent with $\mathcal{S}$, $X$ the set of variables in $A$, and $U$ the set of variables in $X$ that are ground whenever an instance of $A$ is selectable in a goal for $\Pi$. Then, if $U \to X \in D^+$, $A \leftarrow B$ is solution determinate.* □

**Example 6.7** *Consider the program.*

$P(x, v) \leftarrow Q(x, y, u, v) \ \wedge \ R(y, u)$

*where the definitions of $Q/4$ and $R/2$ are as in Example 6.3. Then $\{x\} \to \{x, v\}$ is in the closure of the set of the functional dependencies for $Q(x, y, u, v)$ and $R(y, u)$. Assume that $P/2$ is only selectable when the first argument is ground. Then, it follows from Proposition 6.1 that the clause defining $P/2$ is solution determinate.* □

For a predicate to be solution determinate, not only must each clause in its definition be solution determinate, but also, at most one clause must be applicable in any atomic query with this predicate.

**Definition** Two clauses $A_1 \leftarrow B_1$ and $A_2 \leftarrow B_2$ are *mutually exclusive* if, whenever a selectable atom $A$ unifies with $A_1$ with substitution $\theta_1$ and $A_2$ with substitution $\theta_2$, at most one of the goals $\leftarrow B_1\theta_1$ and $\leftarrow B_2\theta_2$ succeeds.

Thus, if the predicate in $A$ is determinate (head or primitive) then the clauses in the definition of the predicate will be mutually exclusive. However the concepts are not the same.

**Example 6.8** *Consider the following program.*

$P(x, y) \ \leftarrow \ Q(x, y)$
$P(x, y) \ \leftarrow \ R(x, y)$
$Q(1, 1)$
$R(2, 2)$.

*Then with a selection rule that requires the first argument of $P/2$ to be ground, $P/2$ is not head or primitive determinate, but the clauses defining $P/2$ are mutually exclusive.* □

Certain criteria for the detection of mutual exclusion for a pair of clauses in the definition of a predicate, say $P/n$ have been proved in [20]. Excluding the use of cuts, there are two. The basic one corresponds to head determinacy for $P/n$ where $P/n$ is defined by just these two clauses. The second one is an adaptation of primitive determinacy and is more complicated. We only give a simplified version of the second criteria [20, Proposition 3.8].

**Proposition 6.2** *Let $\Pi$ be a program with selection rule $\mathcal{S}$. Then two clauses $A_1 \leftarrow B_1$ and $A_2 \leftarrow B_2$ are mutually exclusive in $\Pi$ (wrt $\mathcal{S}$) if one of the following hold.*

1. *No selectable literal will unify with both $A_1$ and $A_2$.*

2. *The atoms $A_1$ and $A_2$ are identical and the formulas $B_1$ and $B_2$ have the form*

   $B_{11} \wedge L_1 \wedge B_{12}$
   $B_{21} \wedge L_2 \wedge B_{22}$

   *where the $B_{ij}$ $(i, j \in \{1, 2\})$ may be empty formulas; the goals $\leftarrow B_{11}, \leftarrow B_{21}$ are solution determinate; and $L_1, L_2$ are literals with the same predicate such that, whenever a selectable atom unifies with $A_1$ with substitution $\theta$ and $\leftarrow L_1\theta$ succeeds with substitution $\phi$, then $\leftarrow L_2\theta\phi$ also succeeds.*

□

An alternative approach to the definition and detection of mutual exclusivenessis described in [53]. Here clauses $C$ and $C'$ in the definition of a predicate $P$ are mutually exclusive if the subsets of the least Herbrand model that are derived from a bottom-up computation from $C$ and $C'$ are disjoint on the input arguments for $P$.

Sufficient conditions for a predicate to be solution determinate are given below [20, Proposition 3.9].

**Proposition 6.3** *A predicate $P/n$ in a program will be solution determinate if the clauses in the definition of $P/n$ are solution determinate and pairwise mutually exclusive.* □

Thus to check the determinacy of a predicate, each pair of clauses in its definition must be tested, using Proposition 6.2, to see if they are mutually exclusive. Then, if this test succeeds, each clause in the definition must then be checked, using Proposition 6.1, to see if it is solution determinate. The procedure has to iterate, as initially only the determinacy of certain system predicates is given. A complete algorithm based on the methodology of abstract interpretation can now be constructed. The concrete domain is the set of predicates and clauses in the program. The abstract domain has three values *solution determinacy unknown*, *solution determinate* and *not solution determinate*. The algorithm presented in [20] is based on such a procedure. An improved version of the algorithm is described in [53].

We have assumed that the modes of the predicates are already known. However, in practice, it is desirable that the mode, type and determinacy analysis is combined, not only to avoid having more than one analyser, but also because, frequently, there can be considerable interraction between the different abstract domains enabling more accurate results. Such a combined approach is described in [22]. Here, mode, type and the solution determinacy information for a program is obtained by means of a bottom-up abstract interpretation framework.

Clearly, knowledge of the cardinality of a success set for a goal subsumes knowledge of the solution determinacy of a goal. Analysis to find an approximation to the cardinality of a success set has been described in [5]. The analysis applies to full Prolog and is based on an abstract interpretation framework described in [29]. The abstract domain captures not only the cardinality information but also mode and type information, abstractions of built-in atoms, the cut, and termination. Experimental results show that this approach is efficient with a high degree of accuracy.

Analysis of programs to find approximations of the success sets of a goal has been described in [42]. Here an abstract interpretation is performed where the concrete domain is the set of possible variable bindings and the abstract domain is the depth $k$ abstraction of the terms in these bindings. More recently, new methods for obtaining approximations to success sets have been developed [15, 21] and applied to both logic programs and to automated theorem provers. As the success set for a solution determinate goal has at most one element, an approximation to the success set may be useful in the determinacy analysis. Work needs to be done to investigate this and the application of determinacy analysis to automated theorem proving.

A related approach described in [14] performs some analysis that computes, for each predicate, an approximation to the success set where the approximation may include constraints in addition to the equalities. These constraints are then added as tests to each clause in the program. Further analysis is then done to see if the revised predicate definition with these additional tests is primitive determinate.

### Binding Determinacy

The only analysis that we are aware of that has been designed explicitly for binding determinacy is for Reform Prolog [4]. Here, the analysis for parallelisation is hybrid and proactive. At compile-time, a global analyser uses abstract interpretation techniques to find whether a recursive procedure may be safely executed in parallel. The analyser uses abstract domains that provide type, mode and aliasing information concerning the variable bindings and determinacy information indicating where choice points will not be used in the computation. The analyser will mark variables as shared, local to a process, or fragile. If the variable is marked as fragile, then further sharing analysis is performed at run time. Variables that may be shared are analysed to see if they are binding determinate.

## 7 Using Determinacy Information

This section is concerned with techniques for exploiting any determinacy information. It is assumed that this information is either declared or obtained by global analysis at compile time. There have been several proposals. If the determinacy analyser shows that a predicate is determinate for a certain selection rule, then, by reordering the literals in the clauses or adding control annotations, it may be possible to ensure that the implementation respects such a rule. Pruning operators, such as the cut, or conditionals, such as the *If-Then-Else* construct, can be added to the program so as to avoid searching unwanted branches of the SLDNF-tree. By generalising the WAM indexing techniques, primitive determinacy can be exploited directly by the compiler. Finally, in parallel implementations where precise determinacy information is crucial for stream and-parallelisation, run-time tests are used to check for determinacy and then determinate atoms are preferred by the selection rule. Each of these approaches is considered in turn.

## Changing the Selection Rule

Adding control declarations to a program or even reordering the literals in the body of Prolog clauses can be regarded as changing the selection rule.

For instance, the *when* declarations of NU-Prolog [48] are optional program annotations that specify delaying conditions for user-defined predicates and alter the Prolog selection rule. The selection rule of NU-Prolog differs from the left to right selection rule in that it selects the left-most literal that is not delayed by a *when* declaration. The *when* declarations can be automatically generated and added to a program by the NU-Prolog preprocessor called nac [36]. The main intention of nac is to improve efficiency and avoid non-termination for recursive predicates. As a by-product, with the new selection rule, a predicate often becomes head determinate. As explained in [38], having generated and added *when* declarations to the program, nac can also detect which of the predicates are binding determinate.

**Example 7.1** *Consider Example 4.1. With the trivial selection rule, Balanced/1 is not head determinate. However, the nac preprocessor would add wait declarations to this program causing Balanced/2 to be selectable only when the first argument is a non-variable and such that if it is a non-empty list, then the first argument must also be a non-variable. With any selection rule that satisfies these wait declaration, Balanced/2 is head determinate and the predicate Balanced/1 is binding determinate.* □

Predicates that are binding determinate can be executed using dependent and-parallelism without the need for distributed backtracking. This is the rationale behind parallel NU-Prolog [38]. Parallel NU-Prolog is, in fact, an extension to NU-Prolog in which programmers can declare predicates as deterministic with their intended modes by using *lazyDet* declarations. A preprocessor similar to nac then appropriately postpones unifications by replacing the *lazyDet* declarations by *when* declarations. It also detects binding determinacy in the resulting code, hence improving the parallelisation.

## Pruning and Conditionals

The automatic insertion of cuts to prevent unnecessary backtracking in deterministic predicates has been considered by a number of researchers [34, 38]. The Prolog cut relies on the sequential ordering of the literals in the body of a clause as well as the order of the clauses in a definition. As we have not assumed the Prolog computation rule or search order in our definitions, in the following examples the Prolog cut is replaced by the Gödel *one solution* or *bar commit* [27]. These pruning operators are defined for any computation rule and search order.

In [34], the insertion of the Prolog cut corresponds to the addition of a one solution commit. The one solution commit which has the form $\{W\}$ prunes the search tree in such a way as to ensure (at most) one solution of $W$ is found. Suppose $L$ is a literal in the body of the clause and the functional dependencies imply that the set of variables in $L$ that are bound whenever an instance of $L$ is selected, functionally determines the set of all other variables in that instance of $L$. Then $L$ is replaced by $\{L\}$.

**Example 7.2** *Consider again the Example 3.2. Then unnecessary backtracking can be avoided if the preprocessor replaces the definition of BigProject/*1 *by*

$$\mathit{BigProject}(p) \leftarrow \{\mathit{Project}(p,m,e)\} \wedge \mathit{Salary}(m,s) \wedge s > 51000.$$

□

In [38], it is shown how cuts may be added to the program in Example 4.4 to force binding determinacy. We present the corresponding version using the Gödel bar commit in place of the Prolog cut. The Gödel bar commit | is similar to the commit of the concurrent languages and has the semantics of conjunction. Procedurally, it prunes the search tree so that at most one solution for the formula to the left of the bar commit is found and also prunes the other clauses with commits in the same definition. (As Gödel does not enforce the order in which clauses are evaluated, the Gödel bar commit is symmetric.)

**Example 7.3** *Consider the following variation of the program in Example 4.4.*

$Partition(\_, [\,], [\,], [\,])$
$Partition(i, [x|xs], ls, zs) \ \leftarrow$
$\qquad x \geq i \ | \ ls = [x|ys] \ \wedge \ Partition(i, xs, ys, zs)$
$Partition(i, [x|xs], ys, ms) \ \leftarrow$
$\qquad x < i \ | \ ms = [x|zs] \ \wedge \ Partition(i, xs, ys, zs).$

*Suppose that the selection rule is left to right with the additional constraint that* $Partition/4$ *may be selected only when the first and second arguments are ground. Then a choice point will be created whenever the second argument of the query is a non-empty list. However, as the inequality test is the only literal before the commit, this choice-point will be removed before any bindings are made.* □

An alternative method that forces a program to behave deterministically is by using the *If-Then-Else* constructs of NU-Prolog and Gödel. In these languages, a statement of the form

$A \ \leftarrow \ \mathit{If}\ T\ \mathit{Then}\ B_1\ \mathit{Else}\ B_2$

has the same declarative meaning as the pair of clauses

$A \ \leftarrow \ T \ \wedge \ B_1$
$A \ \leftarrow \ \neg T \ \wedge \ B_2$

Thus, if the predicate in $A$ is primitive determinate and defined as above with test goal $\leftarrow T$, it is straightforward to transform it to the *If-Then-Else* form where the test goal is made explicit.

**Example 7.4** *Consider Example 4.4. Then this could be written using the If-Then-Else construct as follows.*

$Partition(\_, [\,], [\,], [\,])$
$Partition(i, [x|xs], y1s, z1s) \ \leftarrow$
$\qquad \mathit{If}\ x \geq i$
$\qquad \mathit{Then}$
$\qquad\quad Partition(i, xs, ys, zs) \ \wedge \ y1s = [x|ys] \ \wedge \ z1s = zs$
$\qquad \mathit{Else}$
$\qquad\quad Partition(i, xs, ys, zs) \ \wedge \ y1s = ys \ \wedge \ z1s = [x|zs].$

*Note that the output bindings for the last two arguments of* $Partition/4$ *are now defined explicitly in the body of the clause.* □

As $T$ occurs negatively in the clause

$A \leftarrow \ \neg T \ \wedge B_2$

every variable in the condition part of an *If-Then-Else* statement (as defined above) must occur in its head since the corresponding argument positions in any selectable atom that unifies with the head must be ground. There is a more general form of the *If-Then-Else* construct that allows for variables in the condition part provided they are local to the *Then* part of the clause. This form of *If-Then-Else* statement has the structure

$A \leftarrow \mathit{If\ Some}\ [\overline{x}]\ T\ \mathit{Then}\ B_1\ \mathit{Else}\ B_2$

where $\overline{x}$ is a sequence of all the variables in $T$ not in $A$. It has the same declarative meaning as the pair of clauses

$A \leftarrow \mathit{Some}\ [\overline{x}]\ (T \wedge B_1)$
$A \leftarrow \neg \mathit{Some}\ [\overline{x}]\ T \wedge B_2.$

The following example taken from [37] illustrates the use of this construct.

**Example 7.5** *Consider the program*

$\mathit{LookUp}(key, v, ls, ls) \leftarrow \mathit{Member}(\mathit{Pair}(key, v), ls)$
$\mathit{LookUp}(key, value, ls, [\mathit{Pair}(key, value)|ls]) \leftarrow \neg \mathit{Member}(\mathit{Pair}(key, \_), ls)$

*which defines the predicate LookUp for looking up a value in an association list. It is assumed that* $\mathit{LookUp}(t_1, t_2, t_3, t_4)$ *is selectable only when* $t_1$ *and* $t_3$ *are ground. The definition of LookUp can be transformed into the statement*

$\mathit{LookUp}(key, value, ls, newls) \leftarrow$
$\quad \mathit{If\ Some}\ [v]\ \mathit{Member}(\mathit{Pair}(key, v), ls)$
$\quad \mathit{Then}$
$\quad\quad value = v \wedge newls = ls$
$\quad \mathit{Else}$
$\quad\quad newls = [\mathit{Pair}(key, value)|ls].$

□

## Indexing

Indexing is a compiler optimisation technique introduced in the WAM [1]. If the evaluation of the clauses in the definition of a predicate was strictly sequential and a choice point created whenever there was more than one clause, then the implementation would be very inefficient in both space and time. It has been observed that programmers usually write code in a directed manner where the first argument in the head of each clause has a distinguishing pattern. The WAM has a technique called *indexing* that optimises clause selection using the top level function of the first argument. There have been a number proposals for extending clause indexing to allow for clause selection based on arguments other than the first, subterms of the arguments, and test goals [26, 50, 52]. We present here a generic methodology for indexing for predicates that are primitive determinate and discuss its relationship with these proposals.

The first step is the *normalisation* of each clause in the definition of a predicate. The head atom is flattened so that its arguments are distinct variables. To preserve the meaning of the clause, equations whose terms have maximum depth 1 are added to the body of the clause.

**Definition** Given an atom or term $A$, we obtain the *flattened form* $(E, A')$ *of* $A$ as follows.

- If $A$ is a variable or constant, then the flattened form of $A$ is $(\{x = A\}, x)$, where $x$ is a variable not in $A$.
- If $A$ is of the form $P(t_1, \ldots, t_n)$ and, for each $i \in \{1, \ldots, n\}$, a flattened form of $t_i$ is $(E_i, t'_i)$, then the flattened form of $A$ is $(E_1 \cup \cdots \cup E_n \cup \{x_1 = t'_1, \ldots, x_n = t'_n\}, P(x_1, \ldots, x_n))$, where $x_1, \ldots, x_n$ are distinct variables that do not occur in $A$ or $t'_1, \ldots, t'_n$.

The *normalised form of a clause* $A \leftarrow B$ is $A' \leftarrow E \wedge B$ where $(E, A')$ is the flattened form of $A$.

**Example 7.6** *Consider Example 4.4. Then the normalised form of the clauses are as follows.*

$C_1$: $\mathit{Partition}(a_1, a_2, a_3, a_4) \leftarrow$
$\quad a_1 = \_ \ \wedge\ a_2 = [\,] \ \wedge\ a_2 = [\,] \ \wedge\ a_2 = [\,]$
$C_2$: $\mathit{Partition}(a_1, a_2, a_3, a_4) \leftarrow$
$\quad a_1 = i \ \wedge\ a_2 = [x|xs] \ \wedge\ a_3 = [x|ys] \ \wedge\ a_4 = zs) \ \wedge$
$\quad x \geq i \ \wedge\ \mathit{Partition}(i, xs, ys, zs)$
$C_3$: $\mathit{Partition}(a_1, a_2, a_3, a_4) \leftarrow$
$\quad a_1 = i \ \wedge\ a_2 = [x|xs] \ \wedge\ a_3 = ys \ \wedge\ a_4 = [x|zs]) \ \wedge$
$\quad x < i \ \wedge\ \mathit{Partition}(i, xs, ys, zs).$

□

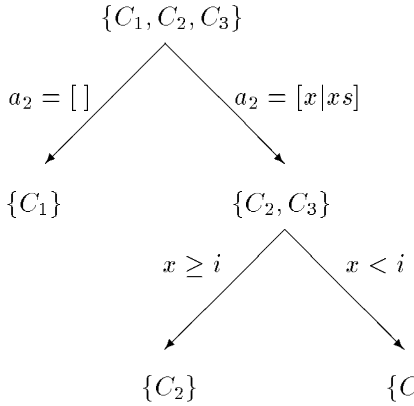
The second step is the construction of an *indexing tree* for a predicate using the normalised forms of the clauses in its definition.

**Definition** Let $\Pi$ be a program with selection rule $\mathcal{S}$ and predicate $P$. An *indexing tree* for $P$ in $\Pi$ is defined as follows. Each node of an indexing tree is labelled by a subset of the normalised forms of the clauses in the definition of $P$ in $\Pi$, called the *candidate clause set* for the node. The edges are labelled by test literals for the normalised clauses.

- The root is labelled by the set of normalised forms of clauses defining $P$.
- Suppose there is a non-leaf node labelled by $\{C_i : i \in I\}$ with $m$ children. Suppose also that, for each $j \in \{1, \ldots, m\}$, the $j$th child is labelled $\{C_i : i \in I_j\}$ and the edge leading to the $j$th child is labelled by a literal $L_j$. Then $I_1, \ldots, I_m$ are distinct non-empty subsets of $I$ such that $I_1 \cup \cdots \cup I_m = I$, and, for each $j \in \{1, \ldots, m\}$, $L_j$ is a test literal wrt $\mathcal{S}$ for all $C_i$, $i \in I_j$.

An indexing tree depends, not only on the clauses in the definition, but also on the choice of test literals used to label the branches. Ideally, these test literals should be chosen so as to minimise the depth of the tree (or, perhaps, minimise the average cost of selecting a clause in the tree [16]). For instance, preference could be given to test literals that cause the nodes with the same parent to have disjoint candidate clause sets. An indexing tree has been defined for any predicate but, for a primitive determinate predicate, it is always possible to extend an indexing tree for it so that each leaf is a singleton set. We call such a tree a *determinate indexing tree*. Note that if $P$ is primitive determinate, once the tests on a branch of a determinate indexing tree have succeeded, no other branches need be considered. If, for a selected atom $A$, the tests on all the branches fail, then the goal $\leftarrow A$ will fail.

**Example 7.7** *We illustrate an indexing tree using the predicate Partition in the Example 4.4.*




$\square$

The *selection tree* defined in [50], differs from an indexing tree, in a number of ways. First, the equality tests in a selection tree only consider constants or top level functions in the predicate's arguments. Secondly, no selection rule is assumed so that mode tests are included as labels for the branches. There are two kinds of non-leaf nodes. The *switch* node does a four-way branch depending on whether the argument in the selected atom is a variable, list, constant or structure and the *hash* node does a multiway branch depending on the value of any constant or top-level function at that position. The selection tree is designed to improve the clause indexing for any predicate, however, if it is known that the predicate is primitive determinate, then, in the implementation of this procedure described in [50], choice points are not created.

One of the problems in deciding that a predicate is primitive determinate is that it may be multi-moded. In [26], mode inferencing techniques are used to determine all the possible modes for each predicate. The program is then transformed into one which contains a new version of each predicate definition for each possible mode. These versions are distinguished by adding the mode as a suffix to their names. Body atoms are renamed according to the mode of their arguments. The basic indexing technique used in [26] is based on a *switching tree* which is similar to the indexing tree defined above.

In [52], a Prolog compiler (TOAM) is described that constructs a *matching tree* for each predicate. Such a tree is similar in structure to the indexing tree defined above but where tests ensure that each leaf is labelled by a singleton set. The leaves are also labelled as cut or noncut. A *cut leaf* is a leaf such that if the tests on the branches to this leaf have succeeded, then the conjunction of tests on each of the branches to its right must fail. (It is assumed that the tree is processed from top to bottom and from left to right.) A choice point is created when the computation reaches a noncut leaf. For primitive determinate predicates, as all the leaves on the matching tree are cut leaves, no choice points will be created.

## Decision Trees

Most work on exploiting the determinacy in parallel implementations has been for Andorra. The basic Andorra-I model selects determinate atoms in preference to other atoms (provided that the behaviour of pruning and built-ins with side-effects is not changed). The determinacy of a predicate depends on the mode of use. However, where predicates are multi-moded, inferring all possible modes for each predicate (as in [26]) can lead to slow compilation. The extra copies of the definition of a predicate for each of its modes and for each mode of use of the atoms in the bodies of the clauses can cause an explosion in the amount of compiled code that is generated. Thus, where accurate determinacy information is required (as is the case with the Andorra-I model), a hybrid system is preferred where static global analysis is combined with run-time local analysis. Both NUA-Prolog [39] and Andorra-I [11] (both of which are languages based on the Andorra model) have systems that use this hybrid methodology for improving the exploitation of and-parallelism in a program. The determinacy information obtained from static analysis is used to construct an indexing tree for each predicate in the program. At run-time, these trees are used in testing the positive literals in the goal to see if they are flat determinate. Those found to be flat determinate are then selected in preference to other atoms and can be executed in parallel with other determinate literals. If these literals were not determinate, distributed backtracking would be required causing a significant overhead for the parallelisation.

NUA-Prolog [39] is an experimental language using the Andorra model and built on the parallel NU-Prolog system, that is hybrid and proactive. It is shown in [39] that generating a complete indexing tree even for head determinacy is NP-hard. To resolve this, assuming the input arguments for an atom are independent, the tree for a predicate of arity, say $n$, is replaced by a sequence of $n$ *decision trees*, one tree for each of its arguments. The candidate clause set at each root node is the set of clauses in the definition of the predicate. The edges leading from the $i$th root are labelled by the test equations for the $i$th argument. Tests labelling edges lower in the $i$th tree may consider equations involving subterms of the $i$th argument or other built-in tests. At run time, given a selected atom, for each argument, the tests are evaluated and a set of candidate clauses at the successful leaf is found. If the intersection of these candidate clause sets is a single clause, then the atom is flat determinate.

A similar hybrid analysis method for the parallel execution of Andorra-I is described by Costa [11]. As in the analysis for NUA-Prolog, a decision tree for each predicate is constructed at compile time. The leaf nodes of the decision tree are labelled *commit* to a named clause (if the candidate clause set is a singleton), or *wait* (if the candidate clause set contains at least two elements). In the latter case, the other argument trees should also be employed. If all argument trees have been considered, and the leaves on the successful branches are all labelled *wait*, then either the atom being considered is not determinate and has to be executed in a non-determinate mode, or the selection of the atom is delayed until it is further instantiated. By ordering the predicate's arguments, the size of the decision tree can be constrained and duplication of work avoided. The actual algorithm also considers the arguments to compound terms and handles built-ins similar to those in the NUA-Prolog algorithm. Note that the decision trees of [11] and [39] were based on the decision trees defined for FCP using the tests in the guard as conditions for the non-leaf nodes [28].

# 8 Discussion

Previously, the study of determinacy and its analysis has been rather ad hoc, primarily depending on the intended application. In particular, definitions of determinacy have assumed specific computation rules and, in some cases, been tailored to fit the actual analysis method. In this paper, we have reformulated the definitions to allow for arbitrary computation rules and search strategies so that the determinacy properties can be inferred using generic analysis techniques such as those based on abstract interpretation. By providing a common framework, the different approaches can be integrated to facilitate significant improvements to the run-time performance of logic programs. Future work will concentrate on producing such an integration and evaluating the performance gains.

Here we have been concerned primarily with definitions of determinacy together with methods for its analysis. In general, we have assumed the programming language to be Prolog or Gödel where determinacy information has to be obtained by analysis. However, if, as in the new logic programming language Mercury [47], determinacy can be declared, then, to a provide a procedure that checks the consistency of the declarations, adapted forms of the analysis techniques can still be utilised.

Languages based on the integration of functional and logic programming paradigms so that both functions and relations may be defined by the user provide a declarative environment in which the intended functionality of many logical expressions may be given explicitly. This functionality information would provide much of the optimising information currently obtained by analysis. Thus programs written in a logic language that allows both functions and relations can reduce the number of places where backtracking is expected and hence, have the potential for compiled code that is more efficient in both space and time than the equivalent program written in a purely relational language.

## Acknowledgements

We are grateful to many colleagues and researchers around the world who have given time to answer many queries concerning their work as well as mailing us copies of their papers and theses containing useful material. In particular, we would like to thank Vitor Santos Costa, Al Roth, Andy Verden, Clive Spenser, Lee Naish, Håkan Millroth, Corin Gurr, Kish Shen, Peter Van Roy, Fergus Henderson, and Thomas Lindgren.

## References

[1] H. Aït-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction.* MIT Press, 1991.

[2] K.R. Apt and I. Luitjes. Verification of logic programs with delay declarations. Technical report, CWI, Amsterdam, 1995. To appear in the Proceedings of the Fourth International Conference on Algebraic Methodology and Software Technology, (AMAST'95).

[3] F. Bergadano, D. Gunetti, and U. Trinchero. The difficulties of learning logic programs with cut. *Journal of Artificial Intelligence Research*, 1:91–107, 1993.

[4] J. Bevemyr, T. Lindgren, and H. Millroth. Reform Prolog: The language and its implementation. In D. S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming,* Budapest, Hungary, pages 283 – 298, 1993.

[5] C. Braem, B. Le Charlier, S. Modart, and P. Van Hentenryck. Cardinality analysis of Prolog. In *Proceedings of the 1995 Symposium on Logic Programming,* Portland, Oregon. MIT Press, 1994.

[6] M. Bruynooghe. Intelligent backtracking for an interpreter of Horn clause logic programs. In B. Dömölki and T. Gergely, editors, *Mathematical Logic in Computer Science*, pages 215–258. North Holland, 1981.

[7] M. Bruynooghe. Intelligent backtracking revisited. In J-L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson.* MIT Press, 1991.

[8] M. Bruynooghe and L.M. Pereira. Deduction revision by intelligent backtracking. In J.A. Campbell, editor, *Implementations of Prolog*, pages 194–215. Ellis Horwood, 1984.

[9] J.-H. Chang and A.M. Despain. Semi-intelligent backtracking of Prolog based on static data dependency analysis. In *International Symposium on Logic Programming*, pages 10–12. IEEE Computer Society, 1985.

[10] V. Santos Costa. *Compile-Time Analysis for the Parallel Execution of Logic Programs in Andorra-I*. PhD thesis, Department of Computer Science, University of Bristol, 1993.

[11] V Santos Costos, D. H. D. Warren, and R. Yang. The Andorra-I engine: A parallel implementation of the basic Andorra model. In K Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming*, pages 825–839. MIT Press, 1991.

[12] P.T. Cox and T. Pietrzykowski. Deduction plans: A basis for intelligent backtracking. *IEEE PAMI*, 3:52–65, 1981.

[13] P. David. Using pivot consistency to decompose and solve functional csps. *Journal of Artificial Intelligence Research*, 2:447–474, 1995.

[14] S. Dawson, C. Ramakrishnan, I. Ramakrishnan, and R. Sekar. Extracting determinacy in logic programs. In D. S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 424–438. MIT Press, 1993.

[15] D.A. De Waal and J.P. Gallagher. The applicability of logic program analysis and transformation to theorem proving. In *Proceedings of the 12th International Conference on Automated Deduction (CADE-12), Nancy*, 1994.

[16] Saumya Debray, Sampath Kannan, and Mukul Paithane. Weighted decision trees. In K. Apt, editor, *Proceedings of the Joint International Conference on Logic Programming,* Washington, USA, pages 654–668, 1992.

[17] S.K. Debray, D. Gudeman, and P. Bigot. Detection and optimisation of suspension-free logic programs. In M. Bruynooghe, editor, *Proceedings of the 1994 International Symposium on Logic Programming*, pages 487–501. MIT Press, 1994.

[18] S.K. Debray and R. Ramakrishnan. Abstract interpretation of logic programs using magic transformations. *Journal of Logic Programming*, 18:149–176, 1994.

[19] S.K. Debray and D.S. Warren. Detection and optimisation of functional computations in Prolog. In *Proceedings of the Third International Conference on Logic Programming*, 1986.

[20] S.K. Debray and D.S. Warren. Functional computations in logic programs. *ACM Transactions on Programming Languages and Systems*, 11:451–481, 1989.

[21] J.P. Gallagher and D.A. de Waal. Regular approximations of logic programs and their uses. Technical Report CSTR-92-06, Department of Computer Science, University of Bristol, 1993.

[22] R. Giacobazzi and L. Ricci. Detecting Determinate Computations by Bottom-up Abstract Interpretation. In *Proceedings of ESOP'92*. Springer-Verlag, 1992.

[23] M.L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, 1993.

[24] C.A. Gurr. Specialising the ground representation in the logic programming language Gödel. In Y. Deville, editor, *Proceedings of LOPSTR 93*. Springer-Verlag, 1993.

[25] S. Haridi and P. Brand. Andorra Prolog: An integration of Prolog and committed choice languages. In *Proceedings of the International Conference on Fifth Generation Computer Systems,* Tokyo, pages 745–754, 1988.

[26] T.J. Hickey and S. Mudambi. Global compilation of Prolog. *The Journal of Logic Programming*, 7(3):193–230, 1989.

[27] P.M. Hill and J.W. Lloyd. *The Gödel Programming Language.* MIT Press, 1994.

[28] S. Kliger and E. Shapiro. A decision tree compilation algorithm for FCP(|,:,?). In R.K. Kowalski, editor, *Proceedings of the Fifth International Conference on Logic Programming,* Seattle, pages 1315–1336. MIT Press, 1988.

[29] B. Le Charlier, S. Rossi, and P. Van Hentenryck. An abstract interpretation framework for almost full Prolog. In *Proceedings of the 1995 Symposium on Logic Programming,* Portland, Oregon. MIT Press, 1995.

[30] J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987.

[31] M. J. Maher and R. Ramakrishnan. Déjà vu in fixpoints of logic programs. Technical report, IBM T J Watson Research Center, 1994. A preliminary version of this paper appeared in NACLP'89.

[32] M.J. Maher. *Equivalences of Logic Programs*, pages 627–658. Computer Science Press, 1988.

[33] M. Marriott, M. Jose Garcia de la Banda, and M. Hermenegildo. Analyzing logic programs with dynamic sceduling. In *ACM Symposium on Principles of Programming Languages*, 1994.

[34] A.O. Mendelzon. Functional dependencies in logic programming. In *Proceedings of International ACM Conference on Very Large Databases,* Stockholm, Sweden, 1985.

[35] H. Millroth. SLDR–resolution: parallelizing structural recursion in logic programs. *Journal of logic programming*, 1995. (to appear).

[36] L. Naish. Automating control for logic programs. *The Journal of Logic Programming,* 3:167–183, 1985.

[37] L. Naish. Negation and quantifiers in NU-Prolog. In E. Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming,* London, pages 624–634. Lecture Notes in Computer Science 225, Springer-Verlag, 1986.

[38] L. Naish. Parallelizing NU-Prolog. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming,* Seattle, pages 1546–1564. MIT Press, 1988.

[39] D. Palmer and L. Naish. NUA-Prolog: An extension to the WAM for parallel Andorra. In K. Furukawa, editor, *Proceedings of the Eighth International Conference on Logic Programming,* Paris, pages 429–442. MIT Press, 1991.

[40] L.M. Pereira and A. Porto. Selective backtracking. In K. L. Clark and S.-A. Tarnlund, editors, *Logic Programming*, pages 107–114. Academic Press, 1982.

[41] D. Sahlin. Determinacy analysis for full Prolog. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 23–30. ACM Press, 1991.

[42] T. Sato and H. Tamaki. Enumeration of success patterns in logic programs. *Theoretical Computer Science*, 34:227–240, 1984.

[43] H. Sawamura and T. Takeshima. Recursive unsolvability of determinacy, solvable cases of determinacy and their applications to Prolog optimisation. In *Proceedings of ILPS'85*, pages 200–207. MIT Press, 1985.

[44] D.C. Sehr. *Automatic Parallelization of Prolog Programs.* PhD thesis, Department of Computer Science, University of Illinois at Urbana Champaign (Report 1577), 1992.

[45] K. Shen. Exploiting dependent and-parallelism in Prolog: the dynamic dependent and-parallel scheme DDAS. In K. Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 717–731. MIT Press, 1992.

[46] K. Shen. Implementing dynamic dependent and-parallelism. In D.S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 167–183. MIT Press, 1993.

[47] Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. Technical Report 95/14, Department of Computer Science, University of Melbourne, 1995. Accepted for publication in The Journal of Logic Programming.

[48] J.A. Thom and J. Zobel. NU-Prolog reference manual, version 1.3. Technical Report TR 86/10, Machine Intelligence Project, Department of Computer Science, University of Melbourne, 1988.

[49] J.D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume 1. Computer Science Press, 1988.

[50] P. Van Roy, B. Demoen, and Y.D. Willems. Improving the execution speed of compiled Prolog with modes, clause selection and determinism. In *TAPSOFT'87*, volume 250 of *Lecture Notes in Computer Science*, pages 111–125. Springer-Verlag, 1987.

[51] H. Xia. *Analyzing Data Dependencies, Detecting AND-Parallelism and Optimizing Backtracking in Prolog Programs.* PhD thesis, Dem Fachbereich 20 (Informatik), Technischen Universiät Berlin, 1989.

[52] Neng–Fa Zhou. Global optimizations for the TOAM. *Journal of Logic Programming*, 15:275–294, 1993.

[53] J. Zobel. *Analysis of Logic Programs.* PhD thesis, Department of Computer Science, Melbourne, 1990.