Detecting Determinacy in Prolog Programs

Andy King[1], Lunjin Lu[2], and Samir Genaim[3]

[1] University of Kent, Canterbury, CT2 7NF, UK
[2] Oakland University, Rochester, MI 48309, USA
[3] Universidad Politécnica de Madrid, Spain

Abstract. In program development it is useful to know that a call to a Prolog program will not inadvertently leave a choice-point on the stack. Determinacy inference has been proposed for solving this problem yet the analysis was found to be wanting in that it could not infer determinacy conditions for programs that contained cuts or applied certain tests to select a clause. This paper shows how to remedy these serious deficiencies. It also addresses the problem of identifying those predicates which can be rewritten in a more deterministic fashion. To this end, a radically new form of determinacy inference is introduced, which is founded on ideas in ccp, that is capable of reasoning about the way bindings imposed by a rightmost goal can make a leftmost goal deterministic.

1 Introduction

Understanding the determinacy behaviour of a logic program is important in program development. To this end, determinacy inference [13] has been proposed as an analysis for inferring conditions on goals that are sufficient to assure determinacy. The key difference between determinacy checking [12, 14, 16] and determinacy inference is that the former verifies that a given goal will generate at most one answer at most once (if it yields any at all) whereas the latter infers, in a single application of the analysis, a class of goals that are deterministic. In addition to ensuring the determinacy of the initial goal, the conditions inferred by [13] ensure the determinacy of each intermediate atomic sub-goal that is invoked whilst solving the initial goal. Therefore, any call that satisfies its determinacy condition cannot (unintentionally) leave a choice-point on the stack.

Determinacy inference is most insightful when it infers a class of calls that differs from what the programmer expects. If the class is smaller than expected, then either the predicate is unintentionally non-deterministic (i.e. buggy), or the analysis is insufficiently precise. If the class is larger than anticipated, then either the predicate possesses properties that the programmer overlooked (i.e. subtle sub-goal interactions that induce determinacy), or it has been coded incorrectly. Alas, determinacy inference was found to be insufficiently precise for programs which used the cut to enforce determinacy. This is because determinacy conditions are derived from conditions, known as mutual exclusion conditions, that are sufficient to ensure that at most one clause of a predicate can derive an answer to a call. These conditions are derived by analysing the constraints that arise in the different clauses of a predicate. Cuts are often introduced so as to avoid applying a test in a clause whose outcome is predetermined by a test in an earlier clause. The absence of such a test prevented a mutual exclusion condition from being inferred. This paper shows that, although this problem may appear insurmountable, that determinacy inference can be elegantly extended to support cuts. The paper also reports how the machinery used to infer mutual exclusion conditions can be refined so as to reason about tests that operate, not on the arguments of a clause, but sub-terms of the arguments. This is also key to inferring accurate mutual exclusion conditions for realistic programs.

As well as enhancing an existing analysis, the paper introduces a new form of determinacy inference. To illustrate the contribution, consider the database:

```
q(a).   q(b).   r(a).
```

and compound goal `q(X), r(X)` which is not dissimilar to a number of goals that we have found in existing programs [19]. The compound goal generates at most one answer, no matter how it is called, due to the way the bindings generated by the rightmost sub-goal constrain the leftmost sub-goal. The analysis of [13] would only infer that the goal is deterministic if called with `X` ground. Yet the vacuous groundness condition of *true* is sufficient for the goal to be determinate (even though it employs backtracking). The value in knowing that the goal is actually determinate is that it alerts the programmer to *where* the program can be improved. If the programmer can verify that the determinacy conditions hold (which is often straightforward) then the goal can be executed under a `once` meta-call without compromising correctness. Equivalently, the goal could be replaced with `q(X), r(X), !`. Either approach would remove any choice-points that remain unexplored and thereby eliminate a performance bug. Alternatively, the programmer might observe that the goal can be reordered to obtain `r(X), q(X)` which will not generate any choice-points at all (though such a reordering might compromise termination).

The new form of determinacy inference reported in this paper can locate these opportunities for optimisation when it is used in conjunction with the existing analysis [13]. The new form of analysis can detect determinacy in the presence of right-to-left flow of bindings; the existing analysis cannot. Hence, any discrepancy between the results of the two analyses identifies a goal that is deterministic, yet could possibly leave choice-points on the stack. Such goals warrant particularly close scrutiny. Without this form of pinpointing, it would be necessary to manually inspect large numbers of non-deterministic predicates.

One technical contribution is in the way the new analysis is realised using suspension inference [4]. The intuition is to add delay declarations to the program so that a goal can only be selected if no more than one clause in the matching predicate can generate an answer. The sub-goals `r(X)` and `q(X)` are thus selected when, respectively, the groundness conditions of *true* and `X` are satisfied. Suspension inference then deduces that the condition *true* is sufficient for the compound goal not to suspend. A correctness result reported in the paper shows that non-suspension conditions can then be reinterpreted as determinacy conditions. In addition to its use in debugging, the analysis has application in the burgeoning area of semi-offline program specialisation (see the discussion in section 6). The paper is organised as follows. Section 2 presents a worked example that illustrates the new form of determinacy inference. Section 3 explains the main correctness result. (The proofs and all the supporting lemmata are all given in [6]). Sections 4 and 5 explain how to support cuts and tests between the sub-terms of arguments. Sections 6 surveys the related work.

2 Worked Example

Since the correctness argument is necessarily theoretical, this section illustrates the key ideas in the new approach to determinacy inference by way of an example:

```
(1) rev([],[]).
(2) rev([X|Xs],Ys) :- rev(Xs,Zs), app(Zs,[X],Ys).

(3) app([],X,X).
(4) app([X|Xs],Ys,[X|Zs]) :- app(Xs,Ys,Zs).
```

2.1 The common ground

The chief novelty in the previous approach to determinacy inference [13] was in the way success patterns for the individual clauses of a predicate were used to infer mutual exclusion conditions for a call to that predicate. The new analysis builds on these mutual exclusion conditions. Such a condition, when satisfied by a call, ensures that no more than one clause of the matching predicate can lead to a successful derivation. To illustrate, consider characterising the success patterns with an argument-size analysis in which size is measured as list-length:

(1) $\texttt{rev}(x_1, x_2)$:- $x_1 = 0$, $x_2 = 0$.
(2) $\texttt{rev}(x_1, x_2)$:- $x_1 \geq 1$, $x_1 = x_2$.

(3) $\texttt{app}(x_1, x_2, x_3)$:- $x_1 = 0$, $x_2 \geq 0$, $x_2 = x_3$.
(4) $\texttt{app}(x_1, x_2, x_3)$:- $x_1 \geq 1$, $x_2 \geq 0$, $x_1 + x_2 = x_3$.

An algorithm that takes, as input, success patterns and produces, as output, mutual exclusion conditions is detailed in our previous paper [13]. Rather than repeating the algorithm, we give the intuition of how rigidity relates to mutual exclusion. If a call $\texttt{rev}(x_1, x_2)$ succeeds with x_1 bound to a rigid list, then so does a new call $\texttt{rev}(x_1, x_2)$ to the new version of the predicate in which x_1 is bound to the length of the list. Hence, if the original clause succeeds with the original clause (1), then so does the new call to the new version of that clause. The presence of the constraint $x_1 = 0$ in the new clause implies that the argument x_1 of the new call was initially zero. The new clause (2) cannot then also succeed because of the constraint $x_1 \geq 1$. Hence the original clause (2) cannot also succeed with the original call. The argument follows in the other direction, hence the rigidity condition x_1 on the original call is sufficient for mutual exclusion. By similar reasoning, the rigidity of x_2 is also sufficient hence the combined condition $x_1 \vee x_2$ is also a mutual exclusion condition. Repeating this argument for $\texttt{app}(x_1, x_2, x_3)$ yields $x_1 \vee (x_2 \wedge x_3)$ [13].

2.2 The problem

The value of mutual exclusion conditions is that if all sub-goals encountered whilst solving a goal satisfy their conditions, then the goal is deterministic [13]. This motivates the application of backward analysis [5] which infers conditions on goals which ensure that a given set of assertions are not violated. By adding assertions that check calls meet their mutual exclusion conditions, the backward analysis will infer conditions on goals that assure determinacy. To illustrate backward analysis, consider clause (2) on the previous page. Since `[X]` is rigid, it is enough for $d_2 = \texttt{Zs} \vee \texttt{Ys}$ to hold for `app(Zs,[X],Ys)` to satisfy its condition. By inspecting the success patterns of `rev` it follows that after the sub-goal `rev(Xs,Zs)` the rigidity property $f_1 = \texttt{Xs} \wedge \texttt{Zs}$ holds. Thus, if the condition $f_1 \rightarrow d_2$ holds before the sub-goal, then $(f_1 \rightarrow d_2) \wedge f_1$, hence d_2, holds after the sub-goal, and thus the mutual exclusion condition for `app(Zs,[X],Ys)` is satisfied. Since $d_1 = \texttt{Xs} \vee \texttt{Zs}$ is the mutual exclusion condition for the `rev(Xs,Zs)`, $d_1 \wedge (f_1 \rightarrow d_2) = (\texttt{Xs} \vee \texttt{Zs})$ guarantees that both sub-goals of the body satisfy their conditions when encountered. To satisfy this condition, it is enough for `[X|Xs]` to be rigid, that is, for `rev` to be called with a rigid first argument.

The astute reader will notice that `rev` is determinate when called with a rigid second argument. To see this, observe that each answer to `rev(Xs, Zs)` will instantiate `Zs` to a list of different size. Now consider executing the compound goal `app(Zs,[X],Ys), rev(Xs, Zs)` with `Ys` rigid. The rigidity of `Ys` ensures that `app(Zs,[X],Ys)` is deterministic and thus `Zs` has at most one solution of fixed size. This, in turn, guarantees that `rev(Xs, Zs)` can yield at most one answer, hence the compound goal is deterministic. (Actually, `rev` gives one answer and then loops in this mode, though this does not compromise determinacy and such goals can always be executed under `once` without compromising correctness). It is therefore disappointing that [13] only discovers one deterministic mode.

2.3 The solution

The two deterministic modes stem from different flows of bindings between the sub-goals. This motivates an analysis that considers different schedulings of sub-goals and selects a sub-goal only when its mutual exclusion condition is satisfied. In effect, mutual exclusion conditions are interpreted as delay conditions like so:

```
delay rev(X, Y) until rigid_list(X) ; rigid_list(Y).
rev([],[]).
rev([X|Xs],Ys) :- rev(Xs,Zs), app(Zs,[X],Ys).
delay app(X, Y, Z) until rigid_list(X) ; (rigid_list(Y) , rigid_list(Z)).
app([],X,X).
app([X|Xs],Ys,[X|Zs]) :- app(Xs,Ys,Zs).
```

The delay declarations, which are reminiscent of Gödel syntax, block `rev` and `app` goals until combinations of their arguments are bound to rigid lists. Suspension inference [4] (which can be considered to be a black-box) is then applicable to this transformation of the original program and, for this derived program, can infer classes of initial goals which cannot lead to suspending states. For these initial goals, the program cannot reduce to a state that only contains suspending sub-goals, that is, sub-goals which violate their mutual exclusion conditions. Since each sub-goal satisfies its condition when executed, the computation is deterministic. Furthermore, executing any such initial goal under left-to-right selection (and we conjecture any selection) will also generate at most one answer [6] — it may not terminate but again this will not undermine determinacy. Applying suspension inference [4] to the above program yields the rigidity conditions of $x_1 \vee x_2$ and $x_1 \vee (x_2 \wedge x_3)$ for $\mathtt{rev}(x_1, x_2)$ and $\mathsf{app}(x_1, x_2, x_3)$ respectively, as desired. Note that the delay conditions are not left in the program after analysis; they are only introduced for the purpose of applying suspension inference.

3 The semantics and the transformation

This section builds toward stating a result which explains how suspension inference can be applied to realise determinacy inference. Rather unusually, the result relates three different semantics. Firstly, a semantics that maps a single call to a Prolog program to a multiset of possible answers. This semantics is rich enough to observe non-determinacy. Secondly, a semantics for success patterns which can express the concept of a mutual exclusion condition. Thirdly, a semantics for ccp that is rich enough for observing non-suspension. In order to express the transformation of a Prolog program, all three semantics have been formulated in terms of a common language syntax and a common computational domain of constraints. This domain is detailed in section 3.1. The semantics (which themselves possess a number of novel features) are presented in sections 3.2 and 3.3. Finally the transformation and the result itself is presented in section 3.4.

3.1 Computational domain of constraints

The computational domain is a set of constraints Con that is ordered by an entailment (implication) relation $\models$. The set Con is assumed to contain equality constraints of the form $\boldsymbol{x} = \boldsymbol{y}$ where $\boldsymbol{x}$ and $\boldsymbol{y}$ are vectors of variables. Syntactically different constraints may entail one another and therefore a relation $\equiv$ is introduced to express equivalence which is defined by $\theta_1 \equiv \theta_2$ iff $\theta_1 \models \theta_2$ and $\theta_2 \models \theta_1$. Since we do not wish to distinguish between constraints that are semantically equivalent but syntactically different, we base our semantics on a domain of equivalence classes $\mathsf{Con}/\!\equiv$ in which a class is denoted by $[\theta]_\equiv$ where θ is a representative member of the class. This domain is ordered by $[\theta_1]_\equiv \models [\theta_2]_\equiv$ iff $\theta_1 \models \theta_2$. The domain is assumed to come equipped with a conjunction operation $[\theta_1]_\equiv \wedge [\theta_2]_\equiv$ and a projection operation $\overline{\exists}_{\boldsymbol{x}}([\theta]_\equiv)$ where $\boldsymbol{x}$ is a vector of variables, both of which are assumed to posses normal algebraic properties [18]. The former conjoins constraints whereas the latter hides information in $[\theta]_\equiv$ that pertains to variables not contained within $\boldsymbol{x}$.

Equality constraints such as $\boldsymbol{x} = \boldsymbol{y}$ provide a way of connecting the actual arguments $\boldsymbol{x}$ of a call with the formal arguments $\boldsymbol{y}$ of the matching procedure. However, equality is doubly useful since, when combined with projection, it provides a way of renaming constraints — an action which is inextricably linked with parameter passing. To systematically substitute each variable in $\boldsymbol{x}$ with its corresponding variable in $\boldsymbol{y}$ within the constraint $[\theta]_\equiv$ it is sufficient to compute $\overline{\exists}_{\boldsymbol{y}}(\overline{\exists}_{\boldsymbol{x}}([\theta]_\equiv) \wedge [\boldsymbol{x} = \boldsymbol{y}]_\equiv)$ provided that $\mathsf{var}(\boldsymbol{x}) \cap \mathsf{var}(\boldsymbol{y}) = \emptyset$. Since renaming is commonplace, we introduce an abbreviation — $\rho_{\boldsymbol{x},\boldsymbol{y}}([\theta]_\equiv)$ — which is defined $\rho_{\boldsymbol{x},\boldsymbol{y}}([\theta]_\equiv) = \overline{\exists}_{\boldsymbol{y}}(\overline{\exists}_{\boldsymbol{x}}([\theta]_\equiv) \wedge [\boldsymbol{x} = \boldsymbol{y}]_\equiv)$ if $\mathsf{var}(\boldsymbol{x}) \cap \mathsf{var}(\boldsymbol{y}) = \emptyset$ and $\rho_{\boldsymbol{x},\boldsymbol{y}}([\theta]_\equiv) = \rho_{\boldsymbol{z},\boldsymbol{y}}(\rho_{\boldsymbol{x},\boldsymbol{z}}([\theta]_\equiv))$ otherwise, where $\boldsymbol{z}$ is a vector of fresh variables. Note that $\rho_{\boldsymbol{x},\boldsymbol{y}}([\theta]_\equiv)$ removes any variables that are not renamed.

Although the domain of equivalence classes $\mathsf{Con}/\!\equiv$ and its associated operators is adequate for the purposes of constructing the three semantics, it is actually simpler to work within a domain of sets of constraints where each set is closed under implication [18]. This domain is $\wp^{\downarrow}(\mathsf{Con}) = \{\Theta \subseteq \mathsf{Con} \mid \downarrow\Theta = \Theta\}$ where $\downarrow\Theta = \{\theta_1 \in \mathsf{Con} \mid \exists \theta_2 \in \Theta . \theta_1 \models \theta_2\}$. The crucial point is that operations over $\mathsf{Con}/\!\equiv$ can be simulated within $\wp^{\downarrow}(\mathsf{Con})$ which has a less complicated structure. For example, consider the conjunction $[\theta_1]_\equiv \wedge [\theta_2]_\equiv$ for some $\theta_1, \theta_2 \in \mathsf{Con}$. If $\Theta_i = \downarrow\{\theta_i\}$ for $i = 1, 2$ then the conjunction can be modeled by just $\Theta_1 \cap \Theta_2$ since $[\theta_1]_\equiv \wedge [\theta_2]_\equiv = [\theta]_\equiv$ iff $\theta \in \Theta_1 \cap \Theta_2$. The projection and renaming operators straightforwardly lift to closed sets of constraints by $\overline{\exists}_{\boldsymbol{x}}(\Theta) = \downarrow\{\overline{\exists}_{\boldsymbol{x}}([\theta]_\equiv) \mid \theta \in \Theta\}$ and $\rho_{\boldsymbol{x},\boldsymbol{y}}(\Theta) = \downarrow\{\rho_{\boldsymbol{x},\boldsymbol{y}}([\theta]_\equiv) \mid \theta \in \Theta\}$.

3.2 Multiset and success set semantics for Prolog programs

To express the transformation that maps a Prolog program into a concurrent program, it is helpful to express both classes of program within the same language. Thus we adopt concurrent constraint programming (ccp) [18] style in which a program P takes the form $P ::= \epsilon | p(\boldsymbol{x}) :- A | P_1.P_2$ where A is an agent that is defined by $A ::= \mathsf{ask}(\Theta) \to A | \mathsf{tell}(\Theta) | A_1, A_2 | \sum_{i=1}^{n} A_i | p(\boldsymbol{x})$ and $\Theta \in \wp^{\downarrow}(\mathsf{Con})$ and throughout $\boldsymbol{x}$ denotes a vector of distinct variable arguments. A Prolog program is merely a program devoid of agents of the form $\mathsf{ask}(\Theta) \to A$. In the concurrent setting, $\mathsf{ask}(\Theta) \to A$ blocks A until the constraint store Φ entails the ask constraint Θ, that is, $\Phi \subseteq \Theta$. In both settings, the agent $\mathsf{tell}(\Theta)$ updates the store from Φ to $\Phi \cap \Theta$ by imposing the constraints Θ. In a Prolog program, the composition operator "," is interpreted as a sequencing operator which induces left-to-right control, whereas in a concurrent program, the same operator is interpreted as parallel composition. For both classes of program $\sum_{i=1}^{n} A_i$ is an explicit choice operator which systematically searches all A_i agents for answers. This represents the most radical departure from classic Prolog but it is a useful construction because, without loss of generality, all predicates can be assumed to be defined with exactly one definition of the form $p(\boldsymbol{x}) :- A$.

The rationale of the multiset semantics is to capture whether an answer does not occur, occurs exactly once or occurs multiply to a given query. In order to make the semantics as simple as possible, the semantics maps a call, not to an arbitrary multiset of answers, but to a restricted class of multisets in which no element occurs more than twice. This restriction still enables non-determinacy to be observed but assures that the equations that define the multiset semantics have a least solution and therefore that the semantics is well-defined.

Before we proceed to the semantics, we need to clarify what is meant by a multiset of answers and how such multisets can be manipulated and written. A multiset of answers is an element in the space $\wp^{\downarrow}(\mathsf{Con}) \rightarrow \{0,1,2\}$, that is, a map which details how many times, if any, that an answer can arise. Henceforth we shall let $\widehat{\mathsf{Con}}$ abbreviate this set of maps. Multiset union over $\widehat{\mathsf{Con}}$ is defined by $M_1 \mathbin{\widehat{\cup}} M_2 = \lambda\Theta.\min(M_1(\Theta) + M_2(\Theta), 2)$. Thus if an element occurs singly in both M_1 and M_2 then it occurs twice in $M_1 \mathbin{\widehat{\cup}} M_2$. However, if it occurs singly in M_1 and twice in M_2 then it occurs twice in $M_1 \mathbin{\widehat{\cup}} M_2$. Henceforth, to simplify the presentation, we write $\lbag\rbag$ for the multiset $\lambda\Theta.0$ and $\lbag\Psi, \Phi, \Phi\rbag$, for instance, for $\lambda\Theta.\mathsf{if}\ \Theta = \Psi\ \mathsf{then}\ 1\ \mathsf{else}\ (\mathsf{if}\ \Theta = \Phi\ \mathsf{then}\ 2\ \mathsf{else}\ 0)$. Furthermore, we adopt multiset comprehensions with the interpretation that the predicate $\Theta \mathrel{\widehat{\in}} M$ succeeds once/twice iff Θ occurs once/twice in M. For example, if $M' = \lbag\Theta, \Theta, \Phi\rbag$ then $\lbag\Psi \cup \Omega \mid \Omega \mathrel{\widehat{\in}} M'\rbag = \lbag\Psi \cup \Theta, \Psi \cup \Theta, \Psi \cup \Phi\rbag$.

The multiset semantics is a mapping of type $\mathsf{Age}_P \rightarrow \wp^{\downarrow}(\mathsf{Con}) \rightarrow \widehat{\mathsf{Con}}$ where Age_P denotes the set of agents that can be constructed from calls to predicates defined within P. The intuition is that, for a given agent $A \in \mathsf{Age}_P$, the multiset semantics maps a closed set Φ to a multiset of closed sets that detail all the possible answers to A given the input Φ.

Definition 1. The mapping $\mathcal{M}_P : \mathsf{Age}_P \rightarrow \wp^{\downarrow}(\mathsf{Con}) \rightarrow \widehat{\mathsf{Con}}$ is the least solution to the following system of recursive equations:

$$
\begin{aligned}
\mathcal{M}_P[\![\mathsf{tell}(\Phi)]\!](\Theta) &= \mathsf{if}\ \Phi \cap \Theta = \emptyset\ \mathsf{then}\ \lbag\rbag\ \mathsf{else}\ \lbag\Phi \cap \Theta\rbag \\
\mathcal{M}_P[\![A_1, A_2]\!](\Theta) &= \widehat{\cup}\lbag\mathcal{M}_P[\![A_2]\!](\Phi) \mid \Phi \mathrel{\widehat{\in}} \mathcal{M}_P[\![A_1]\!](\Theta)\rbag \\
\mathcal{M}_P[\![\textstyle\sum_{i=1}^{n} A_i]\!](\Theta) &= \widehat{\cup}\lbag\mathcal{M}_P[\![A_i]\!](\Theta)\rbag_{i=1}^{n} \\
\mathcal{M}_P[\![p(\boldsymbol{x})]\!](\Theta) &= \lbag\Theta \cap \rho_{\boldsymbol{y},\boldsymbol{x}}(\Phi) \mid \Phi \in \mathcal{M}_P[\![A]\!](\rho_{\boldsymbol{x},\boldsymbol{y}}(\Theta))\rbag \text{ where } p(\boldsymbol{y}) :- A \in P
\end{aligned}
$$

Example 1. Let $\Theta_a = \downarrow\{x = a\}$ and $\Theta_b = \downarrow\{x = b\}$ and consider the program $P = \{q(x) :- \sum_{i=1}^{3} A_i, r(x) :- \mathsf{tell}(\Theta_a), p(x) :- q(x), r(x)\}$ where $A_1 = \mathsf{tell}(\Theta_a)$ and $A_2 = A_3 = \mathsf{tell}(\Theta_b)$ which builds on the example in the introduction. The closed set Con can be used to express unconstrained input, hence:

$$
\begin{array}{lll}
\mathcal{M}_P[\![p(\boldsymbol{x})]\!](\mathsf{Con}) = \lbag\Theta_a\rbag & \mathcal{M}_P[\![p(\boldsymbol{x})]\!](\Theta_a) = \lbag\Theta_a\rbag & \mathcal{M}_P[\![p(\boldsymbol{x})]\!](\Theta_b) = \lbag\rbag \\
\mathcal{M}_P[\![q(\boldsymbol{x})]\!](\mathsf{Con}) = \lbag\Theta_a, \Theta_b, \Theta_b\rbag & \mathcal{M}_P[\![q(\boldsymbol{x})]\!](\Theta_a) = \lbag\Theta_a\rbag & \mathcal{M}_P[\![q(\boldsymbol{x})]\!](\Theta_b) = \lbag\Theta_b, \Theta_b\rbag \\
\mathcal{M}_P[\![r(\boldsymbol{x})]\!](\mathsf{Con}) = \lbag\Theta_a\rbag & \mathcal{M}_P[\![r(\boldsymbol{x})]\!](\Theta_a) = \lbag\Theta_a\rbag & \mathcal{M}_P[\![r(\boldsymbol{x})]\!](\Theta_b) = \lbag\rbag
\end{array}
$$

Although the multiset semantics applies the left-to-right goal selection, it does not apply the top-down clause selection. This does not mean that it cannot observe non-determinacy because, in general, if a call has $n \geq 2$ answers for some input in Prolog then it has at least $m \geq n$ answers for the same input in the multiset semantics as is illustrated below.

Example 2. Consider a program consisting of the clause $p(x) :- x = [\_|y], p(y)$ followed by $p(x) :- x = []$. Because of top-down clause selection, $p(x)$ will loop in Prolog, ie. return no answers, if invoked with x unconstrained. In our presentation, the Prolog program as rendered as a program P consisting of one definition $p(x) :- \sum_{i=1}^{2} A_i$ where $A_1 = \mathsf{tell}(\downarrow\{x = [\_|y]\}), p(y)$ and $A_2 = \mathsf{tell}(\downarrow\{x = []\})$.

Then $\mathcal{M}_P[\![p(\boldsymbol{x})]\!](\mathsf{Con}) = \wr \Theta_i \mid i \geq 0 \wr$ where $\Theta_i = \downarrow\{x = [x_1, \ldots, x_i]\}$ and the semantics observes that $p(x)$ is not determinate when x is unconstrained.

The following success set semantics is the most conventional. It pins down the meaning of a success pattern which underlies the concept of mutual exclusion.

Definition 2. The mapping $\mathcal{S}_P : \mathsf{Age}_P \rightarrow \wp^{\downarrow}(\mathsf{Con})$ is the least solution to the following system of recursive equations:

$$
\begin{aligned}
\mathcal{S}_P[\![\mathsf{tell}(\Phi)]\!] &= \Phi \\
\mathcal{S}_P[\![A_1, A_2]\!] &= \mathcal{S}_P[\![A_1]\!] \cap \mathcal{S}_P[\![A_2]\!] \\
\mathcal{S}_P[\![\textstyle\sum_{i=1}^{n} A_i]\!] &= \cup\{\mathcal{S}_P[\![A_i]\!]\}_{i=1}^{n} \\
\mathcal{S}_P[\![p(\boldsymbol{x})]\!] &= \rho_{\boldsymbol{y},\boldsymbol{x}}(\mathcal{S}_P[\![A]\!]) \text{ where } p(\boldsymbol{y}) :- A \in P
\end{aligned}
$$

3.3 Concurrency semantics for ccp programs

To state the key correctness result, it is necessary to introduce a concurrency semantics for verifying the absence of suspending agents. Whether suspensions manifest themselves or not depends on the quiescent (resting) state of the store, which motivates the following semantics that was inspired by the elegant quiescent state semantics that has been advocated for ccp [18].

Definition 3. The mapping $\mathcal{Q}_P : \mathsf{Age}_P \rightarrow \wp(\wp^{\downarrow}(\mathsf{Con}) \times \{0,1\})$ is the least solution to the following system of recursive equations:

$$
\begin{aligned}
\mathcal{Q}_P[\![\mathsf{ask}(\Phi) \rightarrow A]\!] &= \{\langle \Theta, 1 \rangle \mid \downarrow\Theta = \Theta \wedge \Theta \not\subseteq \Phi\} \cup \{\langle \Theta, b \rangle \in \mathcal{Q}_P[\![A]\!] \mid \Theta \subseteq \Phi\} \\
\mathcal{Q}_P[\![\mathsf{tell}(\Phi)]\!] &= \{\langle \Theta, 0 \rangle \mid \downarrow\Theta = \Theta \wedge \Theta \subseteq \Phi\} \\
\mathcal{Q}_P[\![A_1, A_2]\!] &= \{\langle \Theta, b_1 \vee b_2 \rangle \mid \langle \Theta, b_i \rangle \in \mathcal{Q}_P[\![A_i]\!]\} \\
\mathcal{Q}_P[\![\textstyle\sum_{i=1}^{n} A_i]\!] &= \cup\{\mathcal{Q}_P[\![A_i]\!]\}_{i=1}^{n} \\
\mathcal{Q}_P[\![p(\boldsymbol{x})]\!] &= \{\langle \Theta, b \rangle \mid \downarrow\Theta = \Theta \wedge \langle \Phi, b \rangle \in \mathcal{Q}_P[\![A]\!] \wedge \rho_{\boldsymbol{y},\boldsymbol{x}}(\Phi) = \overline{\exists}_{\boldsymbol{x}}(\Theta)\} \\
&\quad \text{where } p(\boldsymbol{y}) :- A \in P
\end{aligned}
$$

Since quiescent state semantics are not as well known as perhaps they should be within the world of program analysis, we provide some commentary on the recursive equations. The semantics expresses the resting points of an agent [18] and tags each constraint set with either 1 or 0 to indicate whether or not an agent contains a suspending sub-agent (in the latter case the agent has successfully terminated). Consider, for instance, the agent $\mathsf{ask}(\Phi) \rightarrow A$ and the closed set Θ. If $\Theta \not\subseteq \Phi$ then the agent suspends in Θ, and hence quiesces, and thus the pair $\langle \Theta, 1 \rangle$ is included in the set of quiescent states of the agent. Otherwise, if $\Theta \subseteq \Phi$ and $\langle \Theta, b \rangle$ is a quiescent state of A then $\langle \Theta, b \rangle$ is also a quiescent state of $\mathsf{ask}(\Phi) \rightarrow A$. Any set Θ such that $\Theta \subseteq \Phi$ is a succeeding quiescent state of the agent $\mathsf{tell}(\Phi)$. A compound agent A_1, A_2 quiesces under Θ iff Θ is a quiescent set of both A_1 and A_2. The set Θ is tagged as suspending iff either A_1 or A_2 suspend in Θ. The branching agent $\sum_{i=1}^{n} A_i$ inherits quiescent states from each of its sub-agents. Finally, an agent $p(\boldsymbol{x})$ that invokes an agent A via a definition $p(\boldsymbol{y}) :- A$ inherits quiescent states from A by the application of projection and renaming. The intuition is that the variables $\boldsymbol{x}$ and $\boldsymbol{y}$ act as windows on the sets of constraints associated with $p(\boldsymbol{x})$ and A in that they hide information not pertaining to $\boldsymbol{x}$ and $\boldsymbol{y}$ respectively. If these projected sets coincide under renaming and the set for A is quiescent, then the set for $p(\boldsymbol{x})$ is also quiescent.

Example 3. Continuing with the agents A_1, A_2 and A_3 introduced in example 1:

$$\begin{array}{r}
\mathcal{Q}_P[\![\sum_{i=1}^{3} A_i]\!] = \{\langle \Theta, 0\rangle \mid \Theta \subseteq \Theta_a \vee \Theta \subseteq \Theta_b\} \\
\mathcal{Q}_P[\![\mathsf{ask}(\Theta_a) \rightarrow \sum_{i=1}^{3} A_i]\!] = \{\langle \Theta, 1\rangle \mid \Theta \not\subseteq \Theta_a\} \cup \{\langle \Theta, 0\rangle \mid \Theta \subseteq \Theta_a\}
\end{array}$$

The second agent can either suspend with $\Theta \not\subseteq \Theta_a$ or succeed with $\Theta \subseteq \Theta_a$.

3.4 Transforming a Prolog program into a ccp program

Recall that the transformation introduces delay declarations to predicates which suspend a call until its mutual exclusion condition is satisfied. This idea is expressed in a transformation that maps each call $p(\boldsymbol{x})$ to a guarded version $\mathsf{ask}(\Theta) \rightarrow p(\boldsymbol{x})$ where Θ is set of constraints that enforce mutual exclusion. Recall too that the mutual exclusion conditions are derived from success patterns; the success set semantics provides a multiset of possible answers $\wr \Theta_1, \ldots, \Theta_n \wr$ for each call $p(\boldsymbol{x})$ and the mutual exclusion analysis then derives a condition for $p(\boldsymbol{x})$ — a constraint Φ — which, if satisfiable with one Θ_i, is not satisfiable with any another Θ_j. This mutual exclusion property is expressed by the function $\mathsf{mux} : \widehat{\mathsf{Con}} \rightarrow \wp^{\downarrow}(\mathsf{Con})$ which represents the analysis component that derives the mutual exclusion conditions from the success patterns. With these concepts in place, the transformation is defined thus:

Definition 4. Let $\mathcal{S}_P[\![A_i]\!] \subseteq \Theta_i$ and suppose mux satisfies the property that if $\mathsf{mux}(\wr \Theta_i \wr_{i=1}^{n}) = \Phi$ and $\Phi \cap \Theta_i \neq \emptyset$ then $\Phi \cap \Theta_j = \emptyset$ for all $i \neq j$. Then

$$\begin{array}{rr}
 & T[\![\mathsf{tell}(\Phi)]\!] = \mathsf{tell}(\Phi) \\
T[\![P_1.P_2]\!] = T[\![P_1]\!].T[\![P_2]\!] & T[\![A_1, A_2]\!] = T[\![A_1]\!], T[\![A_2]\!] \\
T[\![p(\boldsymbol{x}) :- A]\!] = p(\boldsymbol{x}) :- T[\![A]\!] & T[\![\sum_{i=1}^{n} A_i]\!] = \mathsf{ask}(\mathsf{mux}(\wr \Theta_i \wr_{i=1}^{n})) \rightarrow \sum_{i=1}^{n} T[\![A_i]\!] \\
 & T[\![p(\boldsymbol{x})]\!] = p(\boldsymbol{x})
\end{array}$$

The key result is stated below. It asserts that if $p(\boldsymbol{x})$ is invoked in the transformed program with the constraint Ω imposed, and $p(\boldsymbol{x})$ cannot reduce to a suspending state, then calling $p(\boldsymbol{x})$ in the original program — again with Ω imposed — will produce at most one answer and generate that answer at most once.

Theorem 1. Suppose $\langle \Pi, 1\rangle \notin \mathcal{Q}_{T[\![P]\!]}[\![\mathsf{tell}(\Omega), p(\boldsymbol{x})]\!]$ for all $\Pi \in \wp^{\downarrow}(\mathsf{Con})$. Then $|\mathcal{M}_P[\![p(\boldsymbol{x})]\!](\Omega)| \leq 1$

4 The cut, non-monotonicity and incorrectness

The technique for inferring mutual exclusion conditions [13] is not sensitive to the clause ordering. This does not compromise correctness but, due to the presence of cut, the inferred conditions may be overly strong. To avoid refining the semantics and deriving a new analysis, this section shows how the procedure which computes the mutual exclusion conditions can be refined to accommodate this pruning operator.

The correctness of the analysis is founded on theorem 1 which, in turn, is a consequence of a series of monotonicity results that follows from the definition of the multiset semantics. Alas, the cut is a source of non-monotonicity as is illustrated by the `r` predicate:

```
r(X, Y) :- X = a, !, Y = b.          p(X, Y) :- q(X), r(X, Y).
r(X, Y) :- atomic(X).                q(a).
                                     q(b).
```

If `r(X, Y)` is called under the binding $\{\texttt{X} \mapsto b\}$ then the downward closure of the set of computed answers is $\Theta_1 = \downarrow\{\texttt{X} = b\}$. However, if `r(X, Y)` is called with a more general binding – the empty substitution – the downward closure of the set of answers is $\Theta_2 = \downarrow\{\texttt{X} = a, \texttt{Y} = b\}$. The predicate is non-monotonic because $\Theta_1 \not\subseteq \Theta_2$. The predicate `p` illustrates the significance of monotonicity in that it shows how non-monotonicity can undermine correctness. To see this, consider applying suspension inference in which the `r(X, Y)` goals are never delayed but `q(X)` goals are only selected when `X` is ground. Suspension inference [4] would infer the vacuous condition of *true* for `p(X, Y)` since the compound goal `q(X), r(X, Y)` can be scheduled in right-to-left order without incurring a suspension. However, this inference is unsafe, since the call `p(X, Y)` yields two answers.

One may think that the problem of non-monotonicity is insurmountable but correctness can be recovered by ensuring that the mutual exclusion conditions enforce monotonicity. The observation is that the occurrence of a cut in a clause cannot compromise monotonicity if any calls (typically tests) that arise before the cut are invoked with ground arguments. The idea is thus to strengthen the condition so as to ensure this. For example, `r(X, Y)` is monotonic if `X` is ground. The justification of this tactic is that in top-down clause selection, if a clause containing a cut succeeds, then any following clause is not considered. This behaviour can be modelled by adding negated calls to each clause that follows the cut (which is sound due to groundness [11]). Consider, for example, the following predicate:

```
part([], _, [], []).
part([X | Xs], M, [X | L], G) :- X =< M, !, part(Xs, M, L, G).
part([X | Xs], M, L, [X | G]) :- part(Xs, M, L, G).
```

For the sake of inferring mutual exclusion conditions, observe that the negation of `X =< M` can be inserted into the last clause since this clause is reached only if the test fails. Then, due to the negated test, the second and third clauses are mutually exclusive if the groundness condition $x_1 \wedge x_2$ holds where x_i describes the groundness of the i'th argument. It is not necessary to reason about cut to deduce that the first and second clauses are mutually exclusive if $x_1 \vee x_3$ holds. Likewise, a mutual exclusion condition for the first and third clause is $x_1 \vee x_4$. The cumulative mutual exclusion condition for the whole predicate is therefore

$(x_1 \vee x_3) \wedge (x_1 \vee x_4) \wedge (x_1 \wedge x_2) = x_1 \wedge x_2$. Finally, note that when mechanising this approach, it is not actually necessary to insert the negated calls to deduce the grounding requirements; the negated tests were introduced merely to justify the tactic. Finally, applying this technique to `r` yields the condition x_1.

5 Experimental results

An analyser has been constructed in SICStus 3.10.0 in order to assess the scalability of determinacy inference, study precision issues and investigate whether suspension inference can actually discover bugs. The analyser can be used through a web interface located at http://www.cs.kent.ac.uk/∼amk/detweb.html. The analyser is composed of five components: (1) an argument-size analysis and (2) a depth-k analysis both of which infer success patterns for each clause; (3) an analysis which infers mutual exclusion conditions from the success patterns; (4) a suspension inference which computes determinacy conditions; and (5) a backward analysis [13] that infers determinacy conditions by only considering the left-to-right flow of bindings.

Table 1 summarises the results of four analysis experiments on a range of Prolog programs. The S column is the size of the program measured in the number of predicates. The A, B, C and D columns present the number of deterministic modes inferred for four different types of analysis. To compare against previous work [13], column A details the number of modes inferred using a form of inference that only considers the left-to-right flow of bindings [13] and mutual exclusion conditions derived without consideration of the cut, using a classic depth-k analysis. Column B details the number of deterministic modes inferred using suspension inference [4]. Column C refines this analysis by considering cut in the inference of the mutual exclusion conditions. Column D applies a more refined form of depth-k success pattern analysis to further upgrade the mutual exclusion conditions. The entries marked with a + indicate that the analysis improves on its predecessor in either inferring more modes or inferring more refined modes. Note that a predicate that will contribute 2, say, to the mode count if it is deterministic in 2 modes where both modes do not include the other; this explains why the number of modes can exceed the number of predicates.

The + entries in the B column indicate at least one moding improvement that follows from basing determinacy inference on suspension inference. By scanning the outputs of the analyses for a predicate whose modes differ between the two runs, the programmer can locate a suspicious predicate, ie., a predicate that could silently leave a choice-point on the stack. Such a predicate is determinate, but it is only determinate for some mode because of the way the bindings imposed by a goal on the right constrain a goal on the left. If this were not so, then left-to-right form of determinacy inference [13] would have also deduced this mode. Such a predicate could either be rewritten or executed under once for this particular mode. Note that to apply this form of debugging it is not necessary for the programmer to appreciate how the analysis works; the analysers collaborate and merely suggest where the programmer should focus their effort.

file	S	T	A	B	C	D	%	file	S	T	A	B	C	D	%
aircraft	237	2240	241	241	+241	+241	47	lee-route	13	20	14	14	+14	+14	69
asm	45	160	45	45	+45	+45	57	life	11	20	11	11	+11	11	72
boyer	26	40	33	33	+33	33	19	nand	93	400	93	93	93	+93	60
browse	16	20	16	16	+16	16	62	nbody	48	120	48	48	+48	+49	35
bryant	32	140	33	33	33	+33	75	neural	39	50	50	+52	52	52	43
btree	10	10	10	+10	+14	+18	0	peep	21	120	24	24	24	24	71
chat-80	435	9710	588	588	588	+592	51	press	51	150	56	56	56	+56	64
cp	158	730	239	239	239	+240	50	qplan	44	230	51	51	+52	52	31
circuit	5	10	8	+9	9	9	0	queens	16	20	16	16	+16	+17	37
conman	32	130	32	32	+32	32	78	read	42	130	43	43	43	+43	52
c2	8	0	8	8	+8	8	25	reducer	41	140	42	42	+42	+43	26
cr	6	0	9	+9	9	9	0	robot	26	30	29	+29	+30	+30	38
cw	11	20	13	13	+13	13	9	scc1	17	150	17	17	17	17	88
cs-r	36	370	36	36	36	36	75	sdda	33	70	33	+33	+33	+34	60
dcg	15	20	21	+21	21	21	6	serialize	7	20	9	9	+9	+9	0
dialog	30	20	33	33	+33	+33	33	sieve	6	10	6	6	+6	6	0
disj-r	31	50	31	31	31	31	48	sim	103	960	103	103	103	+105	64
ili	58	220	62	62	+63	63	46	sv	125	2280	131	131	+131	131	60
im	24	50	33	33	+33	33	41	sa	71	120	72	72	+72	72	33
inorder	2	0	2	+2	2	2	0	trs	35	3800	35	35	+35	35	57
kalah	45	70	46	46	46	+46	46	tsp	23	10	23	+23	+27	27	21

Fig. 1. Relative precision experiments: **(A)** using [13]; **(B)** using suspension inference; **(C)** adding cut logic to (B); and **(D)** adding decorated depth-k to (C) using the following name abbreviations c2 = connected2, cp = chat-parser, cr = courses-rules, cw = crypt-wamcc, im = ime-v2-2-1, sa = simple-analyzer and sv = sim-v5-2

The columns C and D quantify how techniques for synthesising the mutual exclusion conditions impact on the overall precision. The + entries in column C confirm that reasoning about cut is important. The D column assesses the impact of applying an enriched form of depth-k analysis in the success pattern analysis that underpins the inference of the mutual exclusion conditions. To illustrate the refinement, consider the merge predicate that arises within mergesort program that can again be found at the above URL. If depth-1 analysis is enriched to track constraints between the variables in a truncated term, then the following success patterns are obtained for the three recursive clauses of merge:

```
merge([A|B],[C|D],[A|E]) :- A < C
merge([A|B],[C|D],[A|E]) :- A = C
merge([A|B],[C|D],[C|E]) :- A > C
```

From these success patterns, it can be deduced that the groundness of the first and second arguments is sufficient for mutual exclusion. This condition cannot be inferred unless the depth-k analysis additionally tracks constraints. The + entries in the D column suggest that this refinement is generally useful. Such improvements may appear incremental, but our work suggests that it is only by combining all these techniques that determinacy inference becomes truly useful.

The T column is a comment on performance since it records the time required to perform all components of an analysis on a 1.4 GHz PC equipped with 640 MByte of memory, running Linux 2.6.15-2. Little variance was observed between the running times of the four analyses — even between A and B which employ different fixpoint engines. The column marked % represents the proportion of the predicates in the program for which a determinacy mode could not be inferred using analysis D. Further investigation is required to determine what fraction of these are actually non-determinate.

Space does not permit us to fully report our work on enhancing argument size analysis for determinacy inference, except to note that techniques devised in termination analysis for inferring norms [2, 7] permit more general mutual exclusion conditions to be inferred. Consider, for example, a predicate that succeeds when its single argument is a list of lists. Such a predicate (named traverse) can be found at the above URL. This predicate traverses the outer list, calling an auxiliary predicate to check that each element is also a list. By using an argument size analysis based on the term-size norm, it is possible to show that the predicate is deterministic if called with a ground argument. When the program is decorated with types, however, type-based norms enable this determinacy condition to be relaxed to a rigid list of rigid lists.

Finally, a number of anomalies were discovered during the experimental evaluation. For example, the `exp(N,X,Y)` predicate [19, Figure 3.5] which realises the function that binds `Y` to `X` raised to the power of `N`, is non-deterministic for $\texttt{N} > 0$ and $\texttt{X} = 0$. These bugs were found by scanning programs for predicates that attempted to realise functions for which no modes could be inferred.

6 Related work

The new analysis reported in this paper also has applications in the new area of semi-online program specialisation [10]. In this scheme, the usual static versus dynamic classification that is used within classic binding-time analysis is refined by adding an additional binding-type semi. As well as always unfolding a static call and never unfolding a dynamic call, the unfolding decision for a semi call is postponed until specialisation time. Determinacy inference fits into this scheme because it provides a way of annotating calls with lightweight unfolding conditions, ie., determinacy conditions. Furthermore, it is not difficult to refine determinacy inference so as to annotate semi calls with conditions that select the clause with which to unfold the call. The net result is more aggressive unfolding. Determinacy inference might also have a role in parallelisation since determinacy can be used as the basis for exploiting a form of and-parallelism [17]. Any discrepancy in the modes inferred with the two forms of determinacy inference pinpoints a predicate that is a good candidate for being rewritten, possibly by reordering goals, so that each call to the atoms in body of the clauses is deterministic, irrespective of the bindings imposed by the other calls. This would open up more opportunities for parallelisation.

Dawson *et al.* [3] extract determinacy information from a logic program by applying a program transformation that simultaneously describes both success pattern constraints and constraints from the calling context. These constraints are then added to each clause without compromising the correctness of the program, so as to reduce backtracking. The authors state that "if the clause conditions of a predicate are pairwise non-unifiable, we infer that the predicate is determinate whenever the input arguments are sufficiently ground". However, to assure determinacy, it is also necessary to ensure that any calls invoked within the body of a clause are themselves deterministic. Ensuring this property leads onto the consideration of various computation rules, and the topic of this paper.

Goal-dependent analysis can be used to ensure that each sub-goal of a given goal cannot succeed with more than one clause of a predicate [12]. The key step is to detect whether two different clauses for a predicate are mutually exclusive with respect to the calling patterns of the predicate. Work on determinacy checking (rather determinacy inference) that is of particular note is that by Braem *et al.* [1] who present an analysis that given a calling mode for a predicate, infers bounds on the number of solutions that can be produced by a call in a given mode. In the context of partial evaluation, Sahlin [16] presents a determinacy analysis that can detect whether if a given goal fail, succeeds once, twice or more times, or whether it possibly loops. Mogensen [14] provides a semantically justified reconstruction of the work of Sahlin [16] based on a denotational semantics for Prolog programs with cut. Quite independently, Le Charlier *et al.* [8] developed a denotational sequence-based abstract interpretation framework for Prolog that can, among other things, be instantiated to obtain Sahlin's [16] determinacy analysis. Interestingly, in partial evaluation, delay declarations are sometimes used to postpone the unfolding of goals until they become sufficiently deterministic [9] which hints at the transformation at the heart of this paper. Further afield, Mercury supports a rich class of determinism categories — det, semidet, multi, nondet and failure — which are used to categorise how many times each mode to a predicate or function can succeed. Signature declarations can also be used in PAN [15] to detect unintended backtracking. Finally, the early literature on functional dependencies is reviewed in [20, Chapter 5].

7 Conclusions

This paper has shown how determinacy inference can be improved by transforming the problem to an analysis problem in concurrency. The paper shows that this approach is flexible enough to handle the cut and accurate enough to locate non-determinacy problems in existing programs.

Acknowledgments We thank John Gallagher, Manual Hermenegildo, Michael Leuschel and Fred Mesnard for discussions on determinacy inference. This work was funded, in part, by NSF grants CCR-0131862 and INT-0327760, the EPSRC grant EP/C015517 and the Royal Society joint project grant 2005/R4-JP.

References

1. C. Braem, B. Le Charlier, S. Modar, and P. Van Hentenryck. Cardinality Analysis of Prolog. In M. Bruynooghe, editor, *International Symposium on Logic Programming*, pages 457–471. MIT Press, 1994.
2. M. Bruynooghe, M. Codish, J. Gallagher, S. Genaim, and W. Vanhoof. Termination Analysis through Combination of Type Based Norms. *ACM Transactions on Programming Languages and Systems*, To appear.
3. S. Dawson, C. R. Ramakrishnan, I. V. Ramakrishnan, and R. C. Sekar. Extracting Determinacy in Logic Programs. In *International Conference on Logic Programming*, pages 424–438. MIT Press, 1993.
4. S. Genaim and A. King. Goal-independent Suspension Analysis for Logic Programs with Dynamic Scheduling. In P. Degano, editor, *European Symposium on Programming*, volume 2618 of *LNCS*, pages 84–98. Springer-Verlag, 2003.
5. A. King and L. Lu. A Backward Analysis for Constraint Logic Programs. *Theory and Practice of Logic Programming*, 2(4–5):517–547, 2002.
6. A. King, L. Lu, and S. Genaim. Determinacy Inference by Suspension Inference. Technical Report 2-05, Computing Laboratory, University of Kent, CT2 7NF, 2005. http://www.cs.kent.ac.uk/pubs/2005/2262/.
7. V. Lagoon and P. J. Stuckey. A Framework for Analysis of Typed Logic Programs. In *International Symposium on Functional and Logic Programming*, volume 2024 of *LNCS*, pages 296–310. Springer-Verlag, 2001.
8. B. Le Charlier, S. Rossi, and P. Van Hentenryck. Sequence-based abstract interpretation of Prolog. *Theory and Practice of Logic Programming*, 2:25–84, 2002.
9. M. Leuschel. Personal Communication on Partial Evaluation, April 2005.
10. M. Leuschel, J. Jørgensen, W. Vanhoof, and M. Bruynooghe. Offline Specialisation in Prolog Using a Hand-Written Compiler Generator. *Theory and Practice of Logic Programming*, 4(1):139–191, 2004.
11. J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
12. P. López-García, F. Bueno, and M. Hermenegildo. Determinacy Analysis for Logic Programs Using Mode and Type information. In *Logic-Based Program Synthesis and Transformation*, volume 3573 of *LNCS*, pages 19–35. Springer-Verlag, 2005.
13. L. Lu and A. King. Determinacy Inference for Logic Programs. In M. Sagiv, editor, *European Symposium on Programming*, volume 3444 of *LNCS*, pages 108–123. Springer-Verlag, 2005.
14. T. Mogensen. A Semantics-Based Determinacy Analysis for Prolog with Cut. In *Ershov Memorial Conference*, volume 1181 of *LNCS*, pages 374–385. Springer-Verlag, 1996.
15. M. Müller, T. Glaß, and K. Stroetmann. Pan - The Prolog Analyzer (Short system description). In *Static Analysis Symposium*, volume 1145 of *LNCS*, pages 387–388. Springer-Verlag, 1996.
16. D. Sahlin. Determinacy Analysis for Full Prolog. In *Partial Evaluation and Semantics Based Program Manipulation*, pages 23–30, 1991. SIGPLAN Notices 26(9).
17. V. Santos Costa, D. H. D. Warren, and R. Yang. Andorra-I Compilation. *New Generation Computing*, 14(1):3–30, 1996.
18. V. A. Saraswat, M. C. Rinard, and P. Panangaden. Semantic Foundations of Concurrent Constraint Programming. In *Principles of Programming Languages*, pages 333–352. ACM Press, 1991.
19. E. Shapiro and L. Sterling. *The Art of Prolog*. MIT Press, 1994.
20. J. Zobel. *Analysis of Logic Programs*. PhD thesis, Department of Computer Science, University of Melbourne, Parkville, Victoria 3052, 1990.