# An Analysis Technique for Extracting Determinacy in Logic Programs*†


Steven Dawson  C.R. Ramakrishnan  I.V. Ramakrishnan  R.C. Sekar

Dept. of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794-4400
{sdawson, cram, ram}@cs.sunysb.edu

Bellcore
445 South Street
Morristown, NJ 07962
sekar@bellcore.com



**Abstract**

Unnecessary backtracking is a principal source of inefficiency in Prolog execution. To avoid the overhead of the general backtracking mechanism, determinate programs should be executed deterministically. Even for programs that are not determinate, failure should be identified early so as to minimize time spent in useless computation. One way to achieve this is by identifying conditions under which a predicate will succeed and checking this condition even before calling the predicate. We present a novel method to infer success conditions of predicates. Unlike previous approaches that rely on *cuts* or use a limited notion of test predicates, we propagate the conditions to detect determinacy, enabling us to handle a much larger class of programs. Since the conditions are propagated explicitly, the power of the method can be readily increased by increasing the expressiveness of these conditions.



*A preliminary version of this paper appeared in ICLP 93 [2].



†Address correspondence to: Prof. I.V. Ramakrishnan, Dept. of Computer Science, SUNY at Stony Brook, Stony Brook, NY 11794-4400.

# 1 Introduction

The exploitation of determinacy in logic programs has been a major focus of efforts to increase the efficiency of Prolog execution. A primary source of inefficiency in Prolog is unnecessary backtracking. Though the general backtracking mechanism is useful to programs that require it, it incurs too much overhead for predicates that essentially compute functions. Introducing *cuts* reduces this overhead, but cut is operational and merely removes choice points that were unnecessary in the first place. An automatic means of exploiting determinacy is a good way to reduce backtracking while preserving the declarative nature of Prolog. This paper presents a novel method to detect determinacy of predicates. The focus of this paper is on the notion of predicate-level determinacy, *i.e.* predicates for which, on any call, at most one clause succeeds. The basic approach is to identify failed computations early, so that backtracking is reduced even in non-determinate predicates. This approach can also be applied to parallel logic programming systems to increase and-parallelism, as discussed in Section 5.

A natural way to infer determinacy of a predicate is to find the conditions under which its clauses succeed. These conditions can be represented as constraints on the arguments of a predicate. If the conditions associated with the clauses of a predicate are mutually exclusive, then we can conclude that the predicate is determinate. Even if the predicate is non-determinate, adding success conditions at the beginning of each clause can promote early failure of unsuccessful computations, making some deep backtracking shallow. An indexing technique that can test these conditions can avoid even shallow backtracking.

In addition to conditions for success of a predicate, its context conditions, *i.e.*, the conditions under which it is called, are important for detecting determinacy. Context conditions are useful in identifying determinacy when a program uses a nondeterminate predicate in a restricted way such that its evaluation is deterministic. Success information originates from facts and base predicates, and percolates up, whereas context information flows down from the top. It is well known that top-down information can be obtained through bottom-up analysis by using transformations such as Magic Templates [11]. We define a similar transformation that enables us to obtain both success and context conditions through a single analysis. Section 2 describes the determinacy analysis method in terms of a program transformation and success condition analysis. The latter is presented in Section 3 and forms the main part of this paper.

Of previous approaches to exploiting determinacy, many rely on the presence of cuts [10, 15]. Such approaches can be viewed as making use only of determinacy information made explicit by the programmer. Others [5, 17, 18] use a limited notion of test predicates, as in the following example:

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

Here determinacy has not been made explicit by the programmer, but must be inferred from the tests $X < Y$ and $X >= Y$. But because these methods do not make such inferences from deeper levels of the program, they are unable to extract hidden, or latent, determinacy. Sato and Tamaki [14] present a technique that extracts determinacy information from deeper levels, but they deal only with programs where determinacy arises from a restricted form of structural equality (term depth abstraction). It is not clear how their method can be extended to handle programs where determinacy arises from other sources, for instance, inequalities or arithmetic constraints. In contrast, the technique presented in this paper propagates constraints explicitly and can uniformly handle a large class of programs. Section 4 illustrates the generality of our method by its applications to programs requiring more than simple structural constraints for extraction of determinacy. In Section 5 we discuss the relationship between our determinacy analysis and other work in greater detail.

# 2 Determinacy Analysis

## 2.1 Notation

The symbols of our language are drawn from three mutually exclusive sets: *variables*, denoted by $X, Y, Z$; *function symbols*, denoted by $a, b, c$; *predicate symbols*, denoted by $p, q, r, s$. The symbols may appear with or without subscripts. *Constants* are 0-ary function symbols. For convenience we use the standard notation for lists; $[H|T]$ for a list with head $H$ and tail $T$, and $[\,]$ for the empty list.

A *term* is a variable, constant, unknown ("_ "), or a compound term $a(t_1, \ldots, t_n)$, where $a$ is a n-ary function symbol, and each $t_i$ is a term. Lists of variables are denoted by symbols $\overline{X}, \overline{Y}, \overline{Z}$. A program is denoted by symbols $\mathcal{P}, \mathcal{P}', \mathcal{Q}$ etc. Each clause in a program is of the form:

$$p(\overline{X}) :- q_1(\overline{X_1}), \ldots, q_n(\overline{X_n}).$$

Note that this form does not restrict the set of programs we consider, since all programs can be readily transformed into this form. We consider purely declarative, negation-free programs, without control features (such as *cut*) or side effects. In the following discussion, clauses are labeled by symbols such as $(\alpha), (\beta)$, etc.; the labels themselves are not a part of the program.

*Constraints* (denoted by $\phi$, $\psi$, or $p^{\#}$, where $p$ is a predicate symbol) are first-order formulae with implicit quantifiers. In a constraint $\phi(\overline{X})$, the variables $\overline{X}$ are universally quantified; other variables are existentially quantified. When there is no ambiguity, $\phi(\overline{X})$ may be written as $\phi$. Logical implication (denoted by '$\Rightarrow$'), defined as usual in terms of substitutions, forms a partial order over the set of constraints.

## 2.2 Overview

To illustrate how success and context conditions are used in inferring determinacy, we consider the following clause:

$$(\alpha): \quad p(\overline{X}) :- q_1(\overline{X_1}), q_2(\overline{X_2}).$$

Let $\phi_{q_1}(\overline{X_1})$ be some *necessary* condition for success of $q_1(\overline{X_1})$; $\phi_{q_1}$ is called the *success condition* for $q_1$. Then the clause

$$(\beta): \;\; p(\overline{X}) :- \phi_{q_1}(\overline{X_1}), q_1(\overline{X_1}), q_2(\overline{X_2}).$$

is (logically) equivalent to clause $\alpha$. Corresponding necessary conditions for other predicates on the right-hand side (rhs) can be introduced similarly.

When the success condition for a predicate (*e.g.* $\phi_{q_1}$) is tested before it is called, many of the calls that would have resulted in failure on the rhs are avoided. Thus, evaluating $p$ through clause ($\beta$) places fewer calls to $q_1$ than evaluating $p$ through ($\alpha$). Furthermore, if the conditions placed on the clauses of a predicate are non-unifiable, then we know that at most one clause is applicable when the input is ground[1]. We can find that unique clause at the time of call and hence not only avoid backtracking, but also avoid placing choice points.

Let $\psi_p$ be a property of predicate $p$ such that $\psi_p(\overline{X})$ is true whenever $p(\overline{X})$ is *called* in some successful derivation; $\psi_p$ is called the *context condition* for $p$. Then the clause

$$(\gamma): \;\; p(\overline{X}) :- \psi_p(\overline{X}), q_1(\overline{X_1}), q_2(\overline{X_2}).$$

is also equivalent to clause $\alpha$.

As previously mentioned, this context information helps in clauses that are not determinate in themselves, but are used in the program in a restricted way such that they become determinate. For instance, let a predicate $p$ be defined by two rules, and let $\phi_1$ and $\phi_2$ be the success conditions on the rhs of each of the two rules. If $\phi_1$ and $\phi_2$ are non-unifiable, we can detect that $p$ is determinate whenever the input is ground. On the other hand, let $\phi_1$ and $\phi_2$ unify to $\phi'$. If $\phi'$ is invalid in every context of $p$, then we can detect that $p$ is used in a determinate manner. Thus we can increase the strength of our method to estimate determinacy at compile time, by using of both success and context information.

Recall that the context condition of a predicate is true whenever that predicate is called from a clause that eventually succeeds. By using a transformation similar to the Magic Templates transformation [11], both success and context conditions can be obtained by bottom-up analysis of the transformed program. Note that the call patterns evaluated by Magic Templates compute the calls that would have been made in a top-down evaluation of the program. Context conditions are different from the call patterns evaluated by Magic Templates, and thus require a different transformation, which is presented below. The analysis method used to compute success and context conditions is described in Section 3.

### 2.3 Derived Program

From a given program $\mathcal{P}$, we obtain a derived program $\mathcal{P}^*$, such that for each predicate $p$ in the original program, we have, in $\mathcal{P}^*$, predicates:

1. $p^{\triangle}$ which computes a necessary condition for success of $p$, and

[1]Henceforth, "ground" means sufficiently instantiated to enable condition testing.

2. $p^{\triangledown}$ which succeeds whenever $p$ is called from a clause that itself succeeds in $\mathcal{P}$.

The success conditions of $p$ and $p^{\triangle}$ are identical. The context conditions of $p$ are the success conditions of $p^{\triangledown}$. The definitions of $p^{\triangle}$ and $p^{\triangledown}$ are given below.

**Definition 2.1 (Success predicates)** *For each clause in $\mathcal{P}$ of the form*

$$p(\overline{X}) :- q_1(\overline{X_1}), \ldots, q_n(\overline{X_n}).$$

*the following clause is in $\mathcal{P}^*$:*

$$p^{\triangle}(\overline{X}) \leftarrow q_1^{\triangle}(\overline{X_1}) \wedge \cdots \wedge q_n^{\triangle}(\overline{X_n})$$

**Definition 2.2 (Context predicates)** *For each user-callable predicate $r$ in $\mathcal{P}$, the clause*

$$r^{\triangledown}(\overline{X}) \leftarrow r^{\triangle}(\overline{X})$$

*is in $\mathcal{P}^*$. And, for each clause in $\mathcal{P}$ of the form*

$$p(\overline{X}) :- q_1(\overline{X_1}), \ldots, q_n(\overline{X_n}).$$

*for each $q_i$, $1 \leq i \leq n$, if $q_i$ is a user-defined predicate, the following clause is in $\mathcal{P}^*$:*

$$q_i^{\triangledown}(\overline{X_i}) \leftarrow p^{\triangledown}(\overline{X}) \wedge q_1^{\triangle}(\overline{X_1}) \wedge \cdots \wedge q_n^{\triangle}(\overline{X_n})$$

To illustrate the construction of a derived program, consider the parser in figure 1 for the LL(2) language $(aa)^n(ab)^n, n \geq 0$. The corresponding derived program is given in Figure 2.

```
p(X1) :- X2 = [], s(X1,X2).
s(X1,X2) :- X1 = [a,a|X3], q(X3,X2).
s(X1,X2) :- X1 = X2.
q(X1,X2) :- s(X1,X3), r(X3,X2).
r(X1,X2) :- X1 = [a,b|X2].
```

Figure 1: Parser for a simple context-free grammar

**Lemma 2.1** *Soundness of $p^{\triangle}$ and $p^{\triangledown}$*

**a.** *Each predicate $p^{\triangle}$ in $\mathcal{P}^*$ succeeds iff the corresponding predicate $p$ in $\mathcal{P}$ succeeds.*

**b.** *Each predicate $p^{\triangledown}$ in $\mathcal{P}^*$ succeeds iff the corresponding predicate $p$ in $\mathcal{P}$ is called from a clause that succeeds in $\mathcal{P}$.*

**Proof:**

**a.** The subset of clauses in $\mathcal{P}^*$ that define $p^{\triangle}$ for each $p$ in $\mathcal{P}$, is obtained via renaming each $p$ in $\mathcal{P}$ by $p^{\triangle}$ . Each $p^{\triangle}$ is equivalent to $p$ and hence defines the necessary and sufficient condition for success of $p$.

**b.** Follows from part (a), above, and operational semantics of Prolog.

■

$$
\begin{array}{rcl}
p^{\triangle}(X_1) & \leftarrow & X_2 = [\,] \wedge s^{\triangle}(X_1, X_2). \\
s^{\triangle}(X_1, X_2) & \leftarrow & X_1 = [a, a\,|X_3] \wedge q^{\triangle}(X_3, X_2). \\
s^{\triangle}(X_1, X_2) & \leftarrow & X_1 = X_2. \\
q^{\triangle}(X_1, X_2) & \leftarrow & s^{\triangle}(X_1, X_3) \wedge r^{\triangle}(X_3, X_2). \\
r^{\triangle}(X_1, X_2) & \leftarrow & X_1 = [a, b\,|X_2]. \\
p^{\triangledown}(X_1) & \leftarrow & p^{\triangle}(X_1). \\
s^{\triangledown}(X_1, X_2) & \leftarrow & X_2 = [\,] \wedge p^{\triangledown}(X_1) \wedge s^{\triangle}(X_1, X_2). \\
s^{\triangledown}(X_1, X_2) & \leftarrow & q^{\triangledown}(X_1, X_3) \wedge s^{\triangle}(X_1, X_2) \wedge r^{\triangle}(X_2, X_3). \\
q^{\triangledown}(X_1, X_2) & \leftarrow & X_3 = [a, a\,|X_1] \wedge s^{\triangledown}(X_3, X_2) \wedge q^{\triangle}(X_1, X_2). \\
r^{\triangledown}(X_1, X_2) & \leftarrow & q^{\triangledown}(X_3, X_2) \wedge s^{\triangle}(X_3, X_1) \wedge r^{\triangle}(X_1, X_2).
\end{array}
$$

Figure 2: Derived program for CFG parser

### 2.4 Determinacy using Success Conditions

Let $q(\overline{X})$ be a predicate in the derived program and $q^{\#}(\overline{X})$ denote a constraint on $\overline{X}$ that is satisfied whenever $q(\overline{X})$ succeeds; *i.e.*, $q^{\#}$ is a necessary condition for success of $q$. Hence, from the definition of $p^{\triangle}$ and $p^{\triangledown}$, $(p^{\triangle})^{\#}$ and $(p^{\triangledown})^{\#}$ are constraints that are satisfied whenever $p$ succeeds and $p$ is called from a successful context, respectively.

**Definition 2.3 (Clause Condition)** *Let $\alpha$ be a clause of the form*

$$p(\overline{X}) :- q_1(\overline{X_1}), \ldots, q_n(\overline{X_n}).$$

*Then the* clause condition *of $\alpha$ is the constraint $\varphi_\alpha$ defined as*

$$\varphi_\alpha = (p^{\triangledown})^{\#} \wedge (\bigwedge_{j=1}^{n} (q_j^{\triangle})^{\#})$$

The constraint $\varphi_\alpha$ defines a necessary condition that $\alpha$ succeeds when $p$ is invoked from a successful context. That is, if $\varphi_\alpha$ is not satisfied, then either the clause fails (one of $(q^{\triangle}{}_j)^{\#}$ is false) or the clause from which $p$ was invoked fails ($(p^{\triangledown})^{\#}$ is false). Hence the constraint $\varphi_\alpha$ can be tested before attempting to satisfy the clause without affecting the success or failure of the program. This is stated formally as follows:

**Proposition 2.2** *Let the program $\mathcal{Q}$ be derived from program $\mathcal{P}$ such that from a clause $\alpha$ in $\mathcal{P}$ of the form*

$$p(\overline{X}) :- q_1(\overline{X_1}), \ldots, q_n(\overline{X_n}).$$

*we derive the following clause in $\mathcal{Q}$:*

$$p(\overline{X}) \leftarrow \varphi_\alpha \wedge q_1(\overline{X_1}) \wedge \ldots \wedge q_n(\overline{X_n})$$

*where $\varphi_\alpha$ is the clause condition of $\alpha$. The programs $\mathcal{P}$ and $\mathcal{Q}$ are equivalent in the sense that any call to an exported predicate yields the same answers in both $\mathcal{P}$ and $\mathcal{Q}$.*

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. Clearly, finding the strongest such conditions for every predicate is undecidable, since this directly reduces to the halting problem. In the next section we develop an analysis technique to find success conditions of predicates that are weak enough to guarantee termination.

## 3 The Analysis Technique

The analysis technique presented in this section finds success conditions of user-defined predicates based on the success conditions of pre-defined predicates. For each predicate $p$ in the program, the analysis determines a constraint $p^\#$ such that, whenever $p(\overline{X})$ succeeds, $p^\#(\overline{X})$ is true. The constraints are first-order formulae that are interpreted over the same domain as the program. Hence, $p(\overline{X}) \Rightarrow p^\#(\overline{X})$, and $p^\#$ is a *necessary condition* for $p$.

We now derive equations to compute $p^\#$ for each user-defined predicate $p$, given $r^\#$ for each pre-defined predicate $r$. The constraint $r^\#$ is fixed *a priori*, depending on $r$. Let $p$ be a predicate in the derived program defined by clauses of the following form:

$$\begin{array}{rcl} p_{\langle 1\rangle}(\overline{X}) & \leftarrow & q_{1,1}(\overline{X}_{1,1}) \wedge \cdots \wedge q_{m_1,1}(\overline{X}_{m_1,1}). \\ \vdots & \vdots & \vdots \\ p_{\langle n\rangle}(\overline{X}) & \leftarrow & q_{1,n}(\overline{X}_{1,n}) \wedge \cdots \wedge q_{m_n,n}(\overline{X}_{m_n,n}). \end{array} \tag{1}$$

Observe that $p$ succeeds only if one of its clauses succeeds, and a clause succeeds only if each of the goals on its rhs succeed. This leads to the following rules that define necessary conditions for satisfaction of each predicate $p$:

$$\begin{array}{rcl} {p_{\langle j\rangle}}^\# & = & \prod_{\overline{X}}(\mathcal{S}_\wedge(\bigwedge_{k=1}^{m_j} (q_{k,j})^\#)) \\ p^\# & = & \mathcal{S}_\vee(\bigvee_{j=1}^{n} {p_{\langle j\rangle}}^\#) \end{array}$$

The operator $\mathcal{S}_\wedge$ performs a conjunction on a set of constraints, $\mathcal{S}_\vee$ performs a disjunction on a set of constraints, and $\prod_{\overline{X}}$ projects a constraint on the variables in $\overline{X}$. Note that in the degenerate case, $\mathcal{S}_\wedge$ is *True* and $\mathcal{S}_\vee$ is *False*.

The above rules can be used directly to compute $p^\#$ if $p$ is nonrecursive. If $p$ is recursively defined, then $p^\#$ is computed by the following fixed point iteration procedure.

$$\begin{array}{rcl} p^{\#,0} & = & \mathit{False} \\ {p_{\langle j\rangle}}^{\#,i} & = & \prod_{\overline{X}}(\mathcal{S}_\wedge(\bigwedge_{k=1}^{m_j} (q_{k,j})^{\#,i-1})) \end{array}$$

$$p^{\#,i} = \mathcal{S}_\vee(\bigvee_{j=1}^{n} p_{\langle j \rangle}{}^{\#,i})$$

For a nonrecursive predicate $q$ (and hence all pre-defined predicates), $q^{\#,i}$ is defined to be $q^{\#}$ for all $i \geq 0$. For a recursive predicate $p$, the limit of the sequence $p^{\#,0}, p^{\#,1}, \ldots$ is defined to be $p^{\#}$.

The purpose of $\mathcal{S}_\wedge$ and $\mathcal{S}_\vee$ is to maintain constraints in a canonical form. This enables detection of fixed points through syntactic identity of constraints. Note that $\mathcal{S}_\wedge$ and $\mathcal{S}_\vee$ are associative, commutative operators. In the following discussion $\mathcal{S}_\wedge$ and $\mathcal{S}_\vee$ are treated as binary operators for simplicity. The computation procedure for $\mathcal{S}_\wedge$, $\mathcal{S}_\vee$, and $\prod$ is defined below.

### 3.1 Constraint simplification

All operations in the constraint simplification procedure maintain constraints in a syntactic form called *standard form.* A constraint is said to be in standard form if it is in disjunctive normal form (DNF), each atom is of the form $(X = t)$, and every universally quantified variable in a conjunction is uniquely and maximally defined.

**Definition 3.1 (Standard Forms)** *A constraint $\phi$ is in* standard form *if it is* True, False *or* $\vee_{i=1}^{n}\psi_i$, *where*

1. *$\psi_i$ is a conjunction;* i.e., $\psi_i = \wedge_{j=1}^{r_i}(X_{ij} = t_{ij})$, *and*
2. *All $X_{ij}$ are universally quantified, and only existentially quantified variables appear in $t_{ij}$.*
3. *In every $\psi_i$, all $X_{ij}$ are distinct.*

Operations $\mathcal{S}_\wedge$, $\mathcal{S}_\vee$, and $\prod$ maintain constraints in *canonical form*, which is a restriction of standard form, and is defined as follows:

**Definition 3.2 (Canonical Forms)** *A constraint $\phi = \vee_{i=1}^{n}\psi_i$ is in* canonical form *if*

1. *$\phi$ is in standard form,*
2. *For every distinct $i$ and $j$, $\psi_i$ is not subsumed by $\psi_j$;* i.e., $\forall_{1\leq i,j\leq n, i\neq j}\exists\sigma\ \psi_i[\sigma] \not\Rightarrow \psi_j[\sigma]$, *and*
3. *In each conjunction $\psi_i = \wedge(X_{ij} = t_{ij})$, if $t_{ij}$ is some variable $Z$, then $Z$ occurs in some $t_{ij}, k \neq j$.*

**Example 3.1** *Consider the four constraints $\phi_1, \phi_2$, $\phi_3$, and $\phi_4$ defined as*

$$\begin{aligned}
\phi_1 &= ((X = [\,]) \wedge (Y = [a])) \vee ((X = [a|U]) \wedge (Y = [b|V])) \\
\phi_2 &= ((X = [\,]) \wedge (Y = [a|X])) \vee ((X = [a|Z]) \wedge (Y = [b|Z])) \\
\phi_3 &= ((X = [a]) \wedge (Y = [b])) \vee ((X = [a|Z]) \wedge (Y = [b|Z])) \\
\phi_4 &= ((X = [\,]) \wedge (Y = [a])) \vee ((X = U) \wedge (Y = [b|V]))
\end{aligned}$$

*$\phi_1$ is in canonical form; $\phi_2$ is not in standard form due to rule 2 of definition 3.1; $\phi_3$ is in standard form but is not in canonical form since the second disjunct subsumes the first. $\phi_4$ is in standard form but is not in canonical form due to rule 3 of definition 3.2.*

In the fixed point iteration procedure, constraints are maintained in canonical form by ensuring that base constraints are in canonical form, and that $\mathcal{S}_\wedge$, $\mathcal{S}_\vee$, and $\prod$ operations return canonical forms when their inputs are in canonical form. These operations are computed according to the rules in figure 3.

$$\mathcal{S}_\vee(\phi_1, \phi_2) \to \mathit{absorb}(\phi_1 \vee \phi_2) \tag{1}$$

$$\mathcal{S}_\wedge(\phi_1, \phi_2) \to \mathit{absorb}(\mathit{product}(\phi_1, \phi_2)) \tag{2}$$

$$\textstyle\prod_{\overline{X}}(\psi_1 \vee \cdots \vee \psi_n) \to \mathit{absorb}(\prod_\wedge(\overline{X}, \psi_1) \vee \cdots \vee \prod_\wedge(\overline{X}, \psi_n)) \tag{3}$$

$$\begin{aligned}&\mathit{absorb}(\psi_1 \vee \cdots \vee \psi_n) \to \\ &\quad \text{if } (\exists i, j \neq i \ \ \psi_i \Rightarrow \psi_j) \text{ then} \\ &\quad \mathit{absorb}(\psi_1 \vee \cdots \vee \psi_{i-1} \vee \psi_{i+1} \vee \cdots \vee \psi_n) \text{ else } \phi\end{aligned} \tag{4}$$

$$\mathit{product}(\phi_1, \phi_2) \to \bigvee \mathit{unify}(\psi_i, \psi_j), \ \text{ for every } \psi_i \in \phi_1, \psi_j \in \phi_2 \tag{5}$$

$$\begin{aligned}&\mathit{unify}((X_1 = t_1 \wedge \cdots \wedge X_n = t_n), (X'_1 = t'_1 \wedge \cdots \wedge X'_m = t'_m)) \to \\ &\quad \mathit{project}(\{X_i \mid 1 \leq i \leq n\} \cup \{X'_j \mid 1 \leq j \leq m\}, S) \\ &\quad \mathit{where} \ \ S = \mathit{mgu}(\{(X_i = t_i) \mid 1 \leq i \leq n\} \cup \{(X'_j = t'_j) \mid 1 \leq j \leq m\})\end{aligned} \tag{6}$$

$$\begin{aligned}&\mathit{project}(\overline{X}, \{(X_i = t_i) \mid 1 \leq i \leq n\}) \to \\ &\quad \textstyle\bigwedge(X_i = t_i), 1 \leq i \leq n, \text{ such that } X_i \in \overline{X}\end{aligned} \tag{7}$$

$$\textstyle\prod_\wedge(\overline{X}, \psi) \to \mathit{truncate}(\mathit{project}(\overline{X}, \mathit{approx}_k(\psi))) \tag{8}$$

$$\begin{aligned}&\mathit{truncate}(\bigwedge_{i=1}^{n}(X_i = t_i)) \to \\ &\quad \textstyle\bigwedge(X_i = t_i), 1 \leq i \leq n, \text{ such that if } t_i \equiv Y \text{ then } \exists j \neq i \ Y \in \mathit{vars}(t_j)\end{aligned} \tag{9}$$

$$\begin{aligned}&\mathit{approx}_k(\bigwedge_{i=1}^{n}(X_i = t_i)) \to \\ &\quad \text{if some } t_i \text{ contains a subterm } t' \equiv f(t'_1, \ldots, t'_m), m > 0 \text{ rooted at depth } k \text{ then} \\ &\qquad \mathit{approx}_k(\textstyle\bigwedge_{i=1}^{n}(X_i = t_i[t' \mapsto f(Z_1, \ldots, Z_m)])), \{Z_1, \ldots, Z_m\} \not\subseteq \mathit{vars}(\bigwedge_{i=1}^{n}(X_i = t_i)) \\ &\qquad \text{else } \{(X_i = t_i) \mid 1 \leq i \leq n\}\end{aligned} \tag{10}$$

Figure 3: Rules defining constraint simplification

In order to illustrate the analysis, we assume just one base predicate: '$=$', the equality predicate. We define $(X = t)^{\#}$ as the constraint $X = t$ converted to canonical form. This means simply that, for each universally quantified variable $Y$ in $t$, $Y$ is replaced by a "new" existentially quantified variable $Z$, and the constraint $Y = Z$ is conjoined with the substituted constraint. For example, if $X$ and $Y$ are universally quantified variables, the constraint $X = Y$ converted to canonical form becomes $(X = Z) \wedge (Y = Z)$, where $Z$ is existentially quantified. Clearly, $X = t$ is equivalent to its canonical form. In order to ensure termination, we limit the depth of terms via $approx_k$ (rule 10), which corresponds to the depth-$k$ abstraction of [14]. In the next section we show how other base predicates can be accommodated in the analysis.

The operator $\mathcal{S}_\vee$ (rule 1) performs the disjunction of two constraints in canonical form, returning a constraint in canonical form. $\mathcal{S}_\vee$ is defined in terms of *absorb* (rule 4), which is responsible for enforcing condition 2 of definition 3.2; *i.e.*, it removes from a DNF any conjunction that is subsumed by another conjunction.

The operator $\mathcal{S}_\wedge$ (rule 2) uses *product* (rule 5) to perform the conjunction of two constraints in canonical form. Since the two inputs are DNFs, *product* performs a kind of "cross product" by unifying each conjunction in the first DNF with each conjunction in the second DNF and returning the disjunction of the unified conjunctions. The heart of the procedure is *unify* (rule 6), which, loosely speaking, finds the most general unifier of two conjunctions in canonical form. Procedure *unify* first forms a set of equations from the two conjunctions, then uses *mgu* to solve the set of equations. The equations are solved according to the standard notion of unification, and the resulting set of equations is in *solved form*, as defined in [9]:

**Definition 3.3 (Solved forms)** *A set of equations is said to be in* solved form *iff it satisfies the following conditions:*

1. *the equations are of the form* $X_i = t_i$, $1 \leq i \leq n$*;*

2. *every variable on the left-hand side of some equation occurs only there.*

The equations in solved form may contain equations with intermediate (existentially quantified) variables on the lhs. In order to preserve canonical forms, such equations are removed by *project* (rule 7).

The operator $\prod$ (rule 3) projects a DNF onto a set of variables. Projection of conjunctions is carried out by $\prod_\wedge$ (rule 8), which first performs any necessary approximation of constraints (in order to guarantee termination of the fixed point iteration procedure). Here, the approximation is done via $approx_k$. The projection on a set of variables is done by procedures *project* and *truncate* (rule 9), which ensures that the projected constraint is in canonical form.

**Example 3.2** *The constraint simplification procedure is illustrated through the computation of the success condition for predicate* $r$, $(r^\triangle)^\#$, *from figure 2. Note that in the first step, the constraint* $(X_1 = [a, b \,|X_2])$ *is converted to canonical form. Also note that the* mgu *step has been omitted, since the set of constraints is already in solved form. A depth-2 approximation is used, albeit imprecisely. Here, "depth" has been taken to mean, "number of non-variable terms at the beginning of the list".*

*The computed success condition for* $r$ *is what one can easily obtain by inspection of the example. The results for the complete example are presented at the end of this section.*

$$
\begin{aligned}
(r^{\triangle})^{\#}(X_1, X_2) &= \prod\nolimits_{\{X_1,X_2\}}(\mathcal{S}_\wedge((X_1 = [a, b\,|X_2])^{\#}, \mathrm{True}))\\
&= \prod\nolimits_{\{X_1,X_2\}}(\mathcal{S}_\wedge(((X_1 = [a, b\,|X_3]) \wedge (X_2 = X_3)), \mathrm{True}))\\
&= \prod\nolimits_{\{X_1,X_2\}}(\mathrm{absorb}(\mathtt{product}(((X_1 = [a, b\,|X_3]) \wedge (X_2 = X_3)), \mathrm{True})))\\
&= \prod\nolimits_{\{X_1,X_2\}}(\mathrm{absorb}(\mathrm{unify}(((X_1 = [a, b\,|X_3]) \wedge (X_2 = X_3)), \mathrm{True})))\\
&= \prod\nolimits_{\{X_1,X_2\}}(\mathrm{absorb}(\mathtt{project}(\{X_1, X_2\}, \{(X_1 = [a, b\,|X_3]), (X_2 = X_3)\})))\\
&= \prod\nolimits_{\{X_1,X_2\}}(\mathrm{absorb}((X_1 = [a, b\,|X_3]) \wedge (X_2 = X_3)))\\
&= \prod\nolimits_{\{X_1,X_2\}}((X_1 = [a, b\,|X_3]) \wedge (X_2 = X_3))\\
&= \mathrm{absorb}(\prod\nolimits_\wedge(\{X_1, X_2\}, (X_1 = [a, b\,|X_3]) \wedge (X_2 = X_3)))\\
&= \mathrm{absorb}(\mathrm{truncate}(\mathtt{project}(\{X_1, X_2\}, \mathit{approx}_2((X_1 = [a, b\,|X_3]) \wedge (X_2 = X_3)))))\\
&= \mathrm{absorb}(\mathrm{truncate}(\mathtt{project}(\{X_1, X_2\}, \{(X_1 = [a, b\,|Z]), (X_2 = X_3)\})))\\
&= \mathrm{absorb}(\mathrm{truncate}((X_1 = [a, b\,|Z]) \wedge (X_2 = X_3)))\\
&= \mathrm{absorb}((X_1 = [a, b\,|Z]))\\
&= (X_1 = [a, b\,|Z])
\end{aligned}
$$

### 3.2 Soundness

We now show that for each predicate $p$ in the program, $p^{\#}$ is a necessary condition[2] for $p$. We first establish the soundness of $\mathcal{S}_\wedge$, $\mathcal{S}_\vee$ and $\prod$, which is then used to prove the soundness of the analysis. Finite computability of $\mathcal{S}_\wedge$, $\mathcal{S}_\vee$ and $\prod$ is proved in the next subsection. Only the main results are presented here. Other lemmas and proofs are in appendix A.

**Theorem 3.1** *The operations $\mathcal{S}_\vee$, $\mathcal{S}_\wedge$ and $\prod$ preserve canonical forms.*

**Lemma 3.2** *$approx_k$ is sound. That is, for any conjunction $\psi$ in canonical form, $\psi \Rightarrow approx_k(\psi)$.*

**Lemma 3.3** *$\mathcal{S}_\vee$ and $\mathcal{S}_\wedge$ are lossless. That is, for all constraints $\phi_1$ and $\phi_2$ in canonical form, $(\phi_1 \vee \phi_2) \Leftrightarrow \mathcal{S}_\vee(\phi_1, \phi_2)$, and $(\phi_1 \wedge \phi_2) \Leftrightarrow \mathcal{S}_\wedge(\phi_1, \phi_2)$.*

**Theorem 3.4** *$\mathcal{S}_\wedge$, $\mathcal{S}_\vee$ and $\prod$ are sound.*

**Proof:** $\mathcal{S}_\wedge$ and $\mathcal{S}_\vee$ are sound since they are lossless (lemma 3.3).

Since *approx*$_k$ is sound (by lemma 3.2), and $\prod_\wedge$ only removes atomic constraints from a conjunction (effectively replacing them by *True*), $\prod_\wedge$ is sound. Furthermore, since *absorb* is lossless (by lemma A.3), it follows that $\prod$ is sound. ∎

[2]Note that the strongest necessary condition corresponds to the least fixed point.

### 3.2.1 Nonrecursive Programs

**Lemma 3.5** *Let $p$ be a predicate as defined before (equation 1 on page 6), and $(q_{k,j})^{\#}$ be a necessary condition for each $q_{k,j}$. Then $p^{\#}(\overline{X}) = \mathcal{S}_{\vee}(\bigvee_{j=1}^{n} \prod_{\overline{X}}(\mathcal{S}_{\wedge}(\bigwedge_{k=1}^{m_j} (q_{k,j})^{\#})))$ is a necessary condition for $p$.*

**Proof:** Since $(q_{k,j})^{\#}(\overline{X}_{k,j})$ is a necessary condition for $q_{k,j}(\overline{X}_{k,j})$ to succeed, we have $(q_{k,j})(\overline{X}_{k,j}) \Rightarrow (q_{k,j})^{\#}(\overline{X}_{k,j})$. Hence, from soundness of $\mathcal{S}_{\wedge}$, $\prod$ and $\mathcal{S}_{\vee}$ (theorem 3.4), it follows that $p^{\#}$ is a necessary condition for $p$. ∎

**Theorem 3.6** *For each predicate $p$ in a nonrecursive program, $p^{\#}$ is a necessary condition for $p$.*

**Proof:** Since the program is nonrecursive, the call graph is a dag. Let the predicates be assigned a level number $l_p$ such that $l_p = 0$ for each base predicate $p$, *i.e.*, those that do not call any other predicate. For all other predicates, $l_p = i$ if $max(l_q) = i - 1$ where $max$ is taken over all predicates $q$ called by $p$. The theorem is proved by induction on level number.

Base case: All level 0 predicates are predefined or are trivially true. The only predefined predicate is '=', and $(t_1 = t_2)^{\#}$ is defined to be exactly $t_1 = t_2$.

Induction Step: Assume that $q^{\#}$ is a necessary condition for all predicates $q$ of level less than $m$. For a level $m$ predicate, say $p$, all calls are to predicates $q_j$ such that $l_{q_j} < m$. By induction hypothesis, ${q_j}^{\#}$ is a necessary condition for each $q_j$. Hence, from lemma 3.5, $p^{\#}$ is a necessary condition for $p$. ∎

### 3.2.2 Recursive Programs

**Theorem 3.7** *If $p^{\#}$ is a fixed point for $p$, then $p^{\#}$ is a necessary condition for $p$.*

**Proof:** For each predicate $p$, let $p$ be defined as in equation 1 on page 6, let $p^{\#}$ be a fixed point for $p$, define

$$p^0(\overline{X}) \leftarrow \mathit{False}$$

and define $p^i, i > 0$ as

$$\begin{array}{rcl} p^i_{\langle 1\rangle}(\overline{X}) & \leftarrow & q^{i-1}_{1,1}(\overline{X}_{1,1}) \wedge \cdots \wedge q^{i-1}_{m_1,1}(\overline{X}_{m_1,1}). \\ \vdots & \vdots & \vdots \\ p^i_{\langle n\rangle}(\overline{X}) & \leftarrow & q^{i-1}_{1,n}(\overline{X}_{1,n}) \wedge \cdots \wedge q^{i-1}_{m_n,n}(\overline{X}_{m_n,n}). \end{array}$$

We first show by induction that, for all $i \geq 0$, $p^{\#}$ is a necessary condition for $p^i$; that is, $p^i \Rightarrow p^{\#}$. Clearly, $p^{\#}$ is a necessary condition for $p^0$, since $\mathit{False} \Rightarrow p^{\#}$. Now, assume for all predicates $q$, $q^i \Rightarrow q^{\#}$, $i < n$. Since the definition of $p^{i+1}$ is non-recursive, we have (by lemma 3.5)

$$\mathcal{S}_{\vee}(\bigvee_{j=1}^{n} \prod\nolimits_{\overline{X}}(\mathcal{S}_{\wedge}(\bigwedge_{k=1}^{m_j} (q_{k,j})^{\#})))$$

is a necessary condition for $p^{i+1}$. And, since $p^{\#}$ is a fix point for $p$, we have by definition

$$p^{\#} = \mathcal{S}_{\vee}(\bigvee_{j=1}^{n} \prod_{\overline{X}}(\mathcal{S}_{\wedge}(\bigwedge_{k=1}^{m_j} (q_{k,j})^{\#})))$$

Therefore, $p^{i+1} \Rightarrow p^{\#}$. Hence, for all $i > 0$, $p^{\#}$ is a necessary condition for $p^i$.

Now, in all computations in which $p$ has recursion depth less than $i$, $p$ and $p^i$ are identical. Thus, it follows that $p^{\#}$ is a necessary condition for $p$. ■

**Corollary 3.8** *For every predicate p, the constraint $p^{\#}$ computed by the fix point iteration procedure is a necessary condition for p.*

### 3.3 Termination

To effectively compute $p^{\#}$ for every predicate $p$, apart from effective procedures to compute $\mathcal{S}_{\wedge}$, $\mathcal{S}_{\vee}$ and $\prod$, we need a function that determines whether two constraints are *equal*, *i.e.*, satisfaction of one implies satisfaction of the other and vice versa. Since the constraints that arise in this analysis are equality constraints among finite trees, we can use resolution [12] to decide this equality. However, the equality test is performed at the end of every iteration in the fix point computation and using resolution at each step would be expensive. Equality checking based on syntax alone would be ideal. Unfortunately, for constraints that arise in this analysis, canonical forms are *not unique*; hence we cannot devise an equality test based on syntax alone.

It is easy to show that $\mathcal{S}_{\wedge}$, $\mathcal{S}_{\vee}$ and $\prod$ are monotonic (with respect to $\Rightarrow$) and thus, for every predicate $p$, the sequence $p^{\#,0}, p^{\#,1}, \ldots$ is a monotonically ascending chain. Hence, a fixed point is reached when $p^{\#,i+1} \Rightarrow p^{\#,i}$. However, the constraints that arise in the analysis (equality constraints on finite trees) do not have the property of *Independence of Negated Constraints* [6], which allows us to check for implication of disjunctions efficiently by considering pairs of conjunctions at a time.

For termination purposes, however, all that is needed is some ordering among the constraints and an effective procedure to test for this ordering. Since all the constraints encountered in our analysis are in standard form, we define such an ordering, $\succeq$, on standard form constraints and show that $\mathcal{S}_{\wedge}$, $\mathcal{S}_{\vee}$ and $\prod$ are monotonic with respect to $\succeq$ also. The ordering $\succeq$ is such that order between disjunctions can be tested by considering pairs of conjunctions at a time, thus yielding an efficient procedure. The order $\succeq$ is defined as follows:

**Definition 3.4 (Syntactic order)** *The* syntactic order $\succeq$ *is a partial order on constraints in standard form, such that*

1. *For all constraints $\phi$,* False $\succeq \phi$ *and* $\phi \succeq$ True.
2. *If $\phi = \vee_{i=1}^{n}\psi_i$ and $\phi' = \vee_{j=1}^{m}\psi'_j$, then $\phi \succeq \phi'$ iff $\forall_i \exists_j \psi_i \succeq \psi'_j$.*
3. *If $\psi$, $\psi'$ are two conjunctions, then $\psi \succeq \psi'$ iff $\psi \Rightarrow \psi'$.*

It is easy to see that whenever $\phi_1 \succeq \phi_2$, $\phi_1 \Rightarrow \phi_2$ [3]. Furthermore, it is easy to derive a procedure to test for $\succeq$ from its definition.

The main results in the proof of termination of the analysis are given below. Other lemmas and proofs appear in appendix A. We begin by establishing the monotonicity of $\mathcal{S}_\wedge$, $\mathcal{S}_\vee$ and $\prod$ and the finiteness of $\prod$. It then follows that the chain $p^{\#,i}$ for every predicate $p$ converges after a finite number of steps. This convergence can be detected by testing for $\succeq$ and hence, for every predicate $p$, $p^{\#}$ can be effectively computed.

**Theorem 3.9** *$\mathcal{S}_\wedge$, $\mathcal{S}_\vee$ and $\prod$ are finitely computable and are monotonic with respect to $\succeq$.*

**Theorem 3.10** *Range of $\prod$ is finite.*

**Lemma 3.11** *For every predicate $p$ in the program, $p^{\#,i} \succeq p^{\#,i+1}$.*

**Proof:** The proof is by induction on $i$. For the base case, $p^{\#,0} \succeq p^{\#,1}$ for every predicate $p$. For nonrecursive predicates, $p^{\#,0} = p^{\#} = p^{\#,1}$, and hence $p^{\#,0} \succeq p^{\#,1}$. For recursive predicates $p^{\#,0} = \mathit{False}$ and $\mathit{False} \succeq \phi$ for every $\phi$. Hence $p^{\#,0} \succeq p^{\#,1}$.

For the induction step, assume that $p^{\#,i} \succeq p^{\#,i+1}$ for all $i < m$. Let $\psi_l$ denote the constraint $\bigvee_{j=1}^{n} \mathcal{S}(\bigwedge_{k=1}^{m_j} (q_{k,j})^{\#,l-1})$. Note that $\psi_{m+1}$ is obtained from $\psi_m$ by replacing $q^{\#,m-1}$ by $q^{\#,m}$. Since $\mathcal{S}_\wedge$, $\mathcal{S}_\vee$ and $\prod$ are monotonic with respect to $\succeq$, and $q^{\#,m-1} \succeq q^{\#,m}$ (by induction hypothesis), $p^{\#,m+1} = \psi_{m+1} \succeq \psi_m = p^{\#,m}$. ∎

**Theorem 3.12** *For every predicate $p$ in the program, $p^{\#}$ can be effectively computed.*

**Proof:** The range of $\prod$ is finite (from theorem 3.10). Since every $p^{\#,i}$ is in the range of $\prod$, the chain $p^{\#,0}, p^{\#,1}, \ldots$ reaches a limit after a finite number of steps. Now, $p^{\#,i}$ is derived from $p^{\#,i-1}$ by finite applications of $\mathcal{S}_\wedge$, $\mathcal{S}_\vee$ and $\prod$, and these three operations are finitely computable (from theorem 3.9), and hence, $p^{\#,i+1}$ is finitely computable from $p^{\#,i}$. Since the chain $p^{\#,i}$ is monotonically ascending with respect to $\succeq$ (from lemma 3.11), the limit can be identified by checking whether $p^{\#,i+1} \succeq p^{\#,i}$. This check can be effectively performed, and hence, the limit can be effectively identified. ∎

## 3.4 An Example

We now illustrate the method by applying it to the example program in Figure 2, using a depth-2 abstraction. The fixed point computation proceeds bottom-up through the call graph, computing fix points for the strongly connected components. Note that $p^{\#,0} = \mathit{False}$ for every predicate $p$ in the derived program. For brevity, only the conditions that have changed at each iteration are given.

[3] Moreover, in all constraint domains which have the independence of negated constraints property, the order $\succeq$ and $\Rightarrow$ coincide.

$$
\begin{aligned}
(r^{\triangle})^{\#,1}(X_1, X_2) &= (X_1 = [a, b \,|Z]) \\
(s^{\triangle})^{\#,1}(X_1, X_2) &= (X_1 = X_2) \\
(q^{\triangle})^{\#,2}(X_1, X_2) &= (X_1 = [a, b \,|Z]) \\
(s^{\triangle})^{\#,3}(X_1, X_2) &= (s^{\triangle})^{\#,1}(X_1, X_2) \vee (X_1 = [a, a \,|Z]) \\
(q^{\triangle})^{\#,4}(X_1, X_2) &= (q^{\triangle})^{\#,2}(X_1, X_2) \vee (X_1 = [a, a \,|Z]) \\
(p^{\triangle})^{\#,1}(X_1) &= (X_1 = [\,]) \vee (X_1 = [a, a \,|Z]) \\
(p^{\triangledown})^{\#,1}(X_1) &= (X_1 = [\,]) \vee (X_1 = [a, a \,|Z]) \\
(s^{\triangledown})^{\#,1}(X_1, X_2) &= (X_1 = [\,] \wedge X_2 = [\,]) \vee (X_1 = [a, a \,|Z] \wedge X_2 = [\,]) \\
(q^{\triangledown})^{\#,2}(X_1, X_2) &= (X_1 = [a, a \,|Z] \wedge X_2 = [\,]) \vee (X_1 = [a, b \,|Z] \wedge X_2 = [\,]). \\
(s^{\triangledown})^{\#,3}(X_1, X_2) &= (s^{\triangledown})^{\#,1}(X_1, X_2) \vee (X_1 = [a, a \,|Z_1] \wedge X_2 = [a, b \,|Z_2]) \vee \\
&\phantom{=} (X_1 = [a, b \,|Z_1] \wedge X_2 = [a, b \,|Z_2]) \\
(q^{\triangledown})^{\#,4}(X_1, X_2) &= (q^{\triangledown})^{\#,2}(X_1, X_2) \vee (X_1 = [a, a \,|Z_1] \wedge X_2 = [a, b \,|Z_2]) \vee \\
&\phantom{=} (X_1 = [a, b \,|Z_1] \wedge X_2 = [a, b \,|Z_2]) \\
(r^{\triangledown})^{\#,1}(X_1, X_2) &= (X_1 = [a, b \,|Z] \wedge X_2 = [\,]) \vee (X_1 = [a, b \,|Z_1] \wedge X_2 = [a, b \,|Z_2])
\end{aligned}
$$

Recall that predicate $s/2$ from figure 1 was non-determinate. By combining the success ($s^{\triangle}$) and context ($s^{\triangledown}$) conditions for $s/2$ and introducing them into the clause heads, we obtain the following determinate version of $s/2$:

```
s([a,a|X],Y) :- q(X,Y).
s([a,b|X],[a,b|X]).
s([],[]).
```

# 4 Generality of the Method

The extent to which determinacy is revealed depends on the tightness of the success conditions we derive. These in turn are directly dependent on the conditions we associate with the base predicates. The stronger the conditions on the base predicates, the greater is the power of the method. In order to utilize these stronger conditions, the inference mechanism should lose as little information as possible. Hence by strengthening the base conditions and the inferencing mechanism, we can enlarge the class of programs for which determinacy can be inferred by our method.

Generality of the method is illustrated through two examples in which depth-$k$ abstraction reveals no determinacy. The first example requires a different abstraction mechanism (a substitute for $approx_k$), but otherwise the analysis method remains unchanged. The second example, whose determinacy arises in part from arithmetic constraints, requires both a different abstraction and a different base constraint solver (a substitute for *mgu*). Furthermore, in order to preserve efficient identification of fixed points, a suitable canonical form for the base constraints is needed, *e.g.*,

```
p(X1) :- X2 = [], s(X1,X2).
s(X1,X2) :- append([a|X3],[a],X1), append([a|X2],[b],X4), s(X3,X4).
s(X1,X2) :- append([c|X3],[a],X1), append([c|X2],[b],X4), s(X3,X4).
s(X1,X2) :- X1 = X2.

append(X1,X2,X3) :- X1 = [], X2 = X3.
append(X1,X2,X3) :- X1 = [X4|X5], X3 = [X4|X6], append(X5,X2,X6).
```

Figure 4: Parser for a context sensitive grammar

the canonical form for linear arithmetic constraints proposed in [7]. Even in the absence of such a canonical form, identification of fixed points can be accomplished using semantic implication checking ($\Rightarrow$), though less efficiently.

### 4.1 Parser for a Context Sensitive Grammar

The program in Figure 4 is a parser for the language $\{ww^R b^n a^n | w \in (a+c)^n, n \geq 0\}$. We analyze this program using a generalization of the structural constraints used to illustrate the method in Section 3. The necessary condition for equality between two terms is that the labels in some $k$ fixed positions in the terms are equal. By "fixed", we mean positions that are defined in the term either with respect to the root, or its rightmost (leftmost) leaf. Note that the depth-$k$ abstraction is a special case of this condition.

In the discussion that follows, a list is represented as $[b_1, b_2, \cdots, e_2, e_1]$, where $b_i$'s are at fixed positions relative to the root and $e_i$'s are fixed relative to the tail. From $X = [a|Y]$, deriving $hd(X) = a$ is a weak necessary condition. Intuitively, we can obtain stronger necessary conditions from the fact that not only is $X$'s head $a$, $X$ and $Y$ have the same last element. The above abstraction permits us to represent and propagate this condition. The derived program is omitted. Of particular interest to the analysis of this program are the necessary conditions derived for $append^{\triangle}$:

$$
\begin{aligned}
(append^{\triangle})^{\#}(X1, X2, X3) \quad = \quad & (X1 = [\,] \wedge X2 = Z \wedge X3 = Z) \vee \\
& (X1 = [X1_1] \wedge (X2 = [\,] \wedge X3 = [X1_1]) \vee \\
& \qquad (X2 = [\ldots, X2_{-1}] \wedge X3 = [X1_1, \ldots, X2_{-1}])) \vee \\
& (X1 = [X1_1, \ldots, X1_{-1}] \wedge (X2 = [\,] \wedge X3 = [X1_1, \ldots, X1_{-1}]) \vee \\
& \qquad (X2 = [\ldots, X2_{-1}] \wedge X3 = [X1_1, \ldots, X2_{-1}]))
\end{aligned}
$$

It is with this stronger condition for equality that we can derive that the second and third arguments of *append* have the same tail, and hence detect determinacy of the predicate $s$. The clause conditions obtained for the three rules defining $s$ are:

$$
\begin{aligned}
\varphi_s^1 &= (X1 = [a, \ldots, a]) \wedge ((X2 = [a, \ldots, b]) \vee (X2 = [c, \ldots, b]) \vee (X2 = [\,])) \\
\varphi_s^2 &= (X1 = [c, \ldots, a]) \wedge ((X2 = [a, \ldots, b]) \vee (X2 = [c, \ldots, b]) \vee (X2 = [\,])) \\
\varphi_s^3 &= (X1 = Z) \wedge (X2 = Z) \wedge ((X2 = [a, \ldots, b]) \vee (X2 = [c, \ldots, b]) \vee (X2 = [\,]))
\end{aligned}
$$

Clearly, the three conditions are mutually exclusive and hence we conclude that the predicate $s$ is determinate.

### 4.2 Quicksort

That the power of our method is not limited to structural equality is illustrated through this example. We show how arithmetic conditions can be handled by the method. The basic idea is that the properties of arithmetic operators should be abstracted as a finite set of necessary conditions. For example, a necessary condition for $X = Y + Z$ is that $((Y \geq 0) \rightarrow (X \geq Z))$. Another way to abstract these operators is to define them precisely up to a certain point, and with inequalities beyond that ceiling. For example the following formula defines a part of the necessary condition for $X = Y + Z$ when arithmetic is interpreted exactly for numbers from 0 to 3, and approximated outside this range.

$$
\begin{aligned}
&((Y = 0) \wedge (X = Z)) \vee \\
&((Y = 1) \wedge (Z = 1) \wedge (X = 2)) \vee \\
&((Y = 2) \wedge (Z = 1) \wedge (X = 3)) \vee \\
&((Y = 3) \wedge (Z = 1) \wedge (X > 3)) \vee \\
&((Y > 3) \wedge (Z = 1) \wedge (X > 3))
\end{aligned}
$$

It is clear that reasoning with such formulae is tractable. It should be noted here that the simplification of formulas done on this domain can be considered as semantic inference, whereas the simplification in the case of structural equality was defined syntactically.

The Quicksort program fragment in Figure 5 employs a simple sorting routine for short lists. This technique is often used in practice to improve the performance of Quicksort. Unfortunately, execution of this program would result in the placing of a choice point for each call to *quicksort*, since the clause that applies cannot be determined until after a call to *length*. Our analysis method is powerful enough to infer the determinacy of *quicksort*.

The analysis of this program is illustrated using a depth-3 approximation combined with the inference rule for addition noted above. We focus here on the conditions for satisfaction of *length*, since it is those conditions that enable the program to be recognized as determinate. The success condition derived for *length* is:

```
quicksort(X1,X2,X3) :- length(X1,X4), X4 =< 3,
                       simplesort(X1,X2,X3).
quicksort(X1,X2,X3) :- X1 = [X4|X5],
                       length(X1,X6), X6 > 3, X10 = [X4|X9],
                       partition(X4,X5,X7,X8),
                       quicksort(X8,X2,X9),
                       quicksort(X7,X10,X3).

length(X1,X2) :- X1 = [], X2 = 0.
length(X1,X2) :- X1 = [_|X3], length(X3,X4), X2 is X4 + 1.
```

Figure 5: Quicksort program fragment

$$
\begin{aligned}
(length^{\triangle})^{\#}(X1, X2) \;=\;& ((X1 = [\,]) \wedge (X2 = 0)) \vee ((X1 = [Z]) \wedge (X2 = 1)) \vee \\
& ((X1 = [Z_1, Z_2]) \wedge (X2 = 2)) \vee ((X1 = [Z_1, Z_2, Z_3]) \wedge (X2 = 3)) \vee \\
& ((X1 = [Z_1, Z_2, Z_3, Z_4 \,|Z_5]) \wedge (X2 > 3))
\end{aligned}
$$

The following are the clause conditions obtained for *quicksort.* For clarity, we show only the part of the conditions that deals with $X1$, since these are the conditions that lead to determinacy.

$$
\begin{aligned}
\varphi_1 &= ((X1 = [\,]) \vee (X1 = [Z]) \vee (X1 = [Z_1, Z_2]) \vee (X1 = [Z_1, Z_2, Z_3])) \\
\varphi_2 &= (X1 = [Z_1, Z_2, Z_3, Z_4 \,|Z_5])
\end{aligned}
$$

# 5 Other Approaches and Applications

## 5.1 Determinacy methods

Other approaches to extracting determinacy in logic programs have been described by Van Roy, Demoen, and Willems [17]; Hickey and Mudambi [5]; and Zhou, Takagi, and Ushijima [18]. While all three approaches address the problem of avoiding backtracking when predicates are determinate, the class of programs in which they can infer determinacy is restricted. In particular, none of these methods is able to infer determinacy in the examples we give, due to the lack of a mechanism for propagating determinacy information (other than mode information). Even in the following simple example, which is based on one in [5], the ability to propagate constraints is necessary for inferring determinacy.

```
p(X,Y) :- less(X,Y), q(X,Y).
p(X,Y) :- geq(X,Y), r(X,Y).
less(a(X1,_),a(X2,_)) :- X1 < X2.
geq(a(X1,_),a(X2,_)) :- X1 >= X2.
```

In contrast to the above approaches, our method of propagating constraints makes the inference of $p$'s determinacy straightforward. Furthermore, our approach of promoting early failure can help avoid deep backtracking, whereas the above approaches can be viewed as optimizing shallow backtracking. Thus, our method generalizes the test condition approach.

The method proposed by Sato and Tamaki [14] does propagate information leading to determinacy detection. However, in their approach, propagation is implicit. Moreover, the information that is propagated is fixed to a restricted form (depth-$k$) of structural equality between terms. Thus, their method is unable to detect determinacy in the CSG and Quicksort programs, or even in the above example. Their approach is based on an item-set construction similar to that used for LR parsers, and whether it can be adapted to propagate more general constraints is unclear.

Another class of approaches to determinacy analysis treats determinacy strictly as a property of predicates; that is, a predicate is either determinate, non-determinate, or perhaps unknown. The inference of determinacy is then viewed as the solution of a set of simultaneous equations over such a 2- or 3-valued set. The earliest examples of this approach are due to Mellish [10] and Sawamura and Takeshima [15]. In both cases, determinacy is inferred only from simple mutual exclusion of clauses, with a heavy reliance on the presence of cuts. The notion of determinacy was generalized to that of functionality in similar methods by Debray and Warren [3], and Giacobazzi and Ricci [4]. Our method can be viewed as a further generalization of these methods, in that the notion of promoting early failure can lead to reduced backtracking even if a predicate is not inferred to be strictly determinate.

### 5.2 Constraint propagation

The introduction of conditions to promote early failure is similar to the refinement optimization of Marriot and Stuckey [8], and the use of constraint propagation is similar to the technique employed by Srivastava and R. Ramakrishnan [16]. However, the method of [8] deals only with bottom-up information, corresponding to our success conditions, and not with context information. The technique of [16] uses a bi-directional constraint propagation technique that incorporates top-down information. However, both methods, which are designed for the CLP paradigm, deal only with constraints explicitly present in programs, whereas our method is more concerned with the inference of conditions from implicit constraints that lead to determinacy.

### 5.3 Applications

Determinacy analysis offers greater efficiency through reduced backtracking not only for sequential Prolog systems, but also for systems that incorporate dependent and-parallelism, such as Andorra-I

[13]. In Andorra-I determinate goals are executed (in and-parallel) before other goals. A determinate goal in Andorra-I is a goal that matches at most one clause head, where the head may be thought of as also including certain builtin goals. Choice points are created for other goals, which may be evaluated in or-parallel. Determinacy analysis can provide additional information for more accurate identification of determinate goals, leading to greater and-parallelism at run time. Furthermore, determinacy analysis may in many cases identify determinate predicates (those for which, on any call, at most one clause succeeds), which can simplify run-time determinacy testing. Of course, the benefits of early failure afforded by success condition analysis apply to parallel as well as sequential Prolog systems.

## 6 Concluding Remarks

Reducing unnecessary backtracking through promotion of early failure is an important optimization that can enhance the performance of logic programs. In this paper we developed an analysis technique for detecting conditions under which the clauses of a predicate will succeed. Checking these conditions before the predicate is called amounts to promoting early failure, thereby avoiding unnecessary backtracking. Furthermore, mutual exclusion of the inferred conditions implies that the predicate is determinate. Our technique is based on representing the success conditions of predicates by constraints and computing them using symbolic constraint solving. This representation and computation unifies and generalizes previous approaches to extracting determinacy.

Note that the stronger the success conditions inferred, the greater is the ability to promote early failure. The strength of the constraints placed on base predicates determines the strength of the inferred conditions, and thus the extent to which determinacy can be extracted. The choice of base constraints is, therefore, an issue of practical importance. On the one hand, one would like to ensure that the base constraints are sufficiently strong to detect the determinacy present. On the other hand, one would like to minimize the computational effort involved inferring determinacy. For example, the analysis of the context-sensitive parser in section 4 required constraints that maintain information about list tails. By contrast, the LL(2) parser in section 2 required only a depth-2 base constraint. It remains open as to how the process of determining appropriate base constraints for the analysis of a given program can be automated. An alternative approach is to apply a strong set of base constraints and inference rules uniformly in the analysis of programs, and use widening techniques [1] to weaken constraints as the analysis proceeds. These are interesting open questions that are worth exploring.

## Acknowledgements

We thank David S. Warren for his encouragement and many fruitful discussions. This work was supported in part by NSF grant CCR-9102159.

## References


[1] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13:103–179, 1992.

[2] 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.

[3] S. Debray and D. S. Warren. Functional computations in logic programs. *ACM Transactions on Programming Languages and Systems*, 11(3):451–481, July 1989.

[4] R. Giacobazzi and L. Ricci. Detecting determinate computations by bottom-up abstract interpretation. In *European Symposium on Programming*, pages 167–181, 1992.

[5] T. Hickey and S. Mudambi. Global compilation of Prolog. *Journal of Logic Programming*, 7:193–230, 1989.

[6] J. Jaffar and M. Maher. Constraint logic programming – a survey. *Journal of Logic Programming*, 10th Anniversary Special Issue 1994.

[7] J. L. Lassez and K. McAloon. Applications of a canonical form for generalized linear constraints. In *International Conference on Fifth Generation Computing Systems*, pages 703–710, 1988.

[8] K. Marriot and P. J. Stuckey. The 3 R's of optimizing constraint logic programs: Refinement, removal and reordering. In *ACM Symposium on Priciples of Programming Languages*, pages 334–344. ACM Press, 1993.

[9] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, April 1982.

[10] C.S. Mellish. Some global optimizations for a Prolog compiler. *Journal of Logic Programming*, 2(1):43–66, 1985.

[11] R. Ramakrishnan. Magic Templates: A spellbinding approach to logic programming. In *Joint International Conference/Symposium on Logic Programming*, pages 140–159. MIT Press, 1988.

[12] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[13] V. Santos Costa, D. H. D. Warren, and R. Yang. Andorra-I: A parallel Prolog system that transparently exploits both and- and or-parallelism. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 83–93, 1991.

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

[15] H. Sawamura and T. Takeshima. Recursive unsolvability of determinacy, solvable cases of determinacy and their applications to Prolog optimization. In *International Logic Programming Symposium*, pages 200–207. MIT Press, 1985.

[16] D. Srivastava and R. Ramakrishnan. Pushing constraint selections. In *ACM Symposium on Priciples of Database Systems*, pages 301–315. ACM Press, 1992.

[17] P. Van Roy, B. Demoen, and Y. D. Willems. Improving the execution speed of compiled Prolog with modes, clause selection and determinism. In *Theory and Practice of Software Development*, pages 111–125, March 1987.

[18] N. Zhou, T. Takagi, and K. Ushijima. A matching tree oriented abstract machine for Prolog. In *International Conference on Logic Programming*, pages 159–173. MIT Press, 1990.

# A Technical appendix

This appendix contains the lemmas and theorems used in the proofs of soundness and termination of the success condition analysis defined by the rules in figure 3 (section 3).

## A.1 Soundness

**Lemma A.1** *$approx_k$ is sound. That is, for any conjunction $\psi$ in canonical form, $\psi \Rightarrow approx_k(\psi)$.*

**Proof:** Let $X = t$ be a constraint in $\psi$. Note that $approx_k$ only replaces some identical subterms with identical variables, and if $t_1$ and $t_2$ are distinct subterms of $t$ that are replaced, they are replaced by distinct variables. Thus, $X = t'$ is a constraint in $approx_k(\psi)$ such that $t$ is an instance of $t'$. So, for every constraint $C$ in $\psi$, there is a constraint $C'$ in $approx_k(\psi)$ such that $C \Rightarrow C'$. Hence, $approx_k$ is sound. ■

**Lemma A.2** unify *is lossless and preserves canonical forms. That is, for all conjunctions $\psi_1$ and $\psi_2$ in canonical form, $\psi_1 \wedge \psi_2 \Leftrightarrow$ unify$(\psi_1, \psi_2)$, and* unify$(\psi_1, \psi_2)$ *is in canonical form.*

**Proof:** To show that *unify* is lossless, let $\psi_1 = (X_1 = t_1 \wedge \cdots \wedge X_n = t_n)$ and $\psi_2 = (X'_1 = t'_1 \wedge \cdots \wedge X'_m = t'_m)$, and let $S = \{(X''_k = t''_k) \mid 1 \leq k \leq l\} = mgu(\{(X_i = t_i) \mid 1 \leq i \leq n\} \cup \{(X'_j = t'_j) \mid 1 \leq j \leq m\})$. By definition of *mgu*, we have

$$\text{A substitution } \sigma \text{ satisfies } \psi_1 \wedge \psi_2 \text{ iff } \sigma \text{ satisfies } S. \tag{A.2.1}$$

and, since $S$ is in solved form (definition 3.3),

$$\{X''_1, \ldots, X''_l\} \cap vars(\{t''_1, \ldots, t''_l\}) = \emptyset \tag{A.2.2}$$

Now, let $\overline{X} = \{X_i \mid 1 \leq i \leq n\} \cup \{X'_j \mid 1 \leq j \leq m\}$, $\psi' = project(\overline{X}, S)$ and $\overline{t} = \{t''_i \mid 1 \leq k \leq l, X''_k \in \overline{X}\}$. If a substitution $\sigma$ satisfies $\psi'$, it also satisfies $S$, since $(\{X''_1, \ldots, X''_l\} - \overline{X}) \cap vars(\{t''_1, \ldots, t''_l\}) =$

$\emptyset$ and any variable in $\{t''_1, \ldots, t''_l\} - \bar{t}$ that also occurs in $\bar{t}$ can take the substitution specified by $\sigma$. Thus,

$$\text{A substitution } \sigma \text{ satisfies } \psi' \text{ iff } \sigma \text{ satisfies } S. \tag{A.2.3}$$

Hence, from (A.2.1), (A.2.2), (A.2.3), and the definition of *unify*, it follows that

$$\psi_1 \wedge \psi_2 \Leftrightarrow \mathit{unify}(\psi_1, \psi_2).$$

To prove that *unify* preserves canonical forms, it suffices to show that $\mathit{unify}(\psi_1, \psi_2)$ is in standard form, and that any rhs of a constraint in $\mathit{unify}(\psi_1, \psi_2)$ that is a variable occurs in the rhs of some other constraint. That $\mathit{unify}(\psi_1, \psi_2)$ is in standard form follows from (A.2.2) and the fact that $S$ is in solved form. Now, suppose some constraint $X = t$ in $\mathit{unify}(\psi_1, \psi_2)$ is such that $t$ is a variable occurring nowhere else in $\mathit{unify}(\psi_1, \psi_2)$. Without loss of generality, let $X = t'$ be a constraint in $\psi_1$. Since, by assumption, $\psi_1$ is in canonical form, if $t'$ is a variable then $t'$ must occur in the rhs of some other constraint. Thus, it is easy to construct a substitution for $X = t$ that does not satisfy $X = t'$, contradicting the losslessness of *unify*. ■

**Lemma A.3** absorb *is lossless and preserves canonical forms. That is, for every constraint $\phi = (\psi_1 \vee \cdots \vee \psi_n)$, such that each $\psi_i$ is in canonical form, $\phi \Leftrightarrow$* absorb($\phi$)*, and* absorb($\phi$) *is in canonical form.*

**Proof:** Let $\phi = (\psi_1 \vee \cdots \vee \psi_n)$. Since the set of conjunctions in $\mathit{absorb}(\phi)$ is a subset of the set of conjunctions in $\phi$, $\mathit{absorb}(\phi) \Rightarrow \phi$. To show $\phi \Rightarrow \mathit{absorb}(\phi)$, consider two cases: (1) there is no $\psi_i \in \phi$ such that $\psi_i \Rightarrow \psi_j, i \neq j$; (2) for some distinct $i$ and $j$, $\psi_i \Rightarrow \psi_j$. In case (1) $\mathit{absorb}(\phi) = \phi$, so clearly $\phi \Rightarrow \mathit{absorb}(\phi)$. In case (2), $\psi_i \Rightarrow \psi_j$. Thus, $(\psi_1 \vee \cdots \vee \psi_n) \Rightarrow (\psi_1 \vee \psi_{i-1} \vee \cdots \vee \psi_{i+1} \vee \psi_n)$, so $\phi \Rightarrow \mathit{absorb}(\phi)$.

To show that $\mathit{absorb}(\phi)$ is in canonical form, observe that $\mathit{absorb}(\phi)$ is in standard form because $\phi$ is in standard form. Since each conjunction $\psi_i$ of $\phi$ is in canonical form, it suffices to show that in $\mathit{absorb}(\phi)$, no conjunction is implied by another. By definition of *absorb*, no two distinct conjunctions $\psi_i$ and $\psi_j$ in $\mathit{absorb}(\phi)$ are such that $\psi_i \Rightarrow \psi_j$. ■

**Lemma A.4** product *is lossless and preserves standard forms. That is, for all constraints $\phi_1$ and $\phi_2$ in canonical form, $(\phi_1 \wedge \phi_2) \Leftrightarrow$* product($\phi_1, \phi_2$)*;* product($\phi_1, \phi_2$) *is in standard form and every conjunction in* product($\phi_1, \phi_2$) *is in canonical form.*

**Proof:** Losslessness of *product* follows directly from losslessness of *unify* (lemma A.2) and distributivity of $\wedge$ over $\vee$.

Since $\mathit{product}(\phi_1, \phi_2)$ is in DNF, and *unify* preserves canonical forms (lemma A.2), it follows that $\mathit{product}(\phi_1, \phi_2)$ is in standard form, and every conjunction in $\mathit{product}(\phi_1, \phi_2)$ is in canonical form. ■

**Theorem A.5** $\mathcal{S}_\vee$ *and* $\mathcal{S}_\wedge$ *are lossless. That is, for all constraints $\phi_1$ and $\phi_2$ in canonical form, $(\phi_1 \vee \phi_2) \Leftrightarrow \mathcal{S}_\vee(\phi_1, \phi_2)$, and $(\phi_1 \wedge \phi_2) \Leftrightarrow \mathcal{S}_\wedge(\phi_1, \phi_2)$.*

**Proof:** Losslessness of $\mathcal{S}_\vee$ follows directly from losslessness of *absorb* (lemma A.3). Losslessness of $\mathcal{S}_\wedge$ follows from losslessness of *product* (lemma A.4) and losslessness of *absorb* (lemma A.3). ■

**Lemma A.6** $\prod_\wedge$ *preserves canonical forms.*

**Proof:** Since $approx_k$ only replaces some subterms of right-hand sides of constraints with new existential variables, it preserves standard forms. Thus, $\prod_\wedge$ preserves standard forms. Furthermore, $\prod_\wedge$ removes (via *truncate*) constraints of the form $X = Y$, where $Y$ is a variable occurring nowhere else in the conjunction. Hence, $\prod_\wedge$ preserves canonical forms. ■

**Theorem A.7** *The operations $\mathcal{S}_\vee$, $\mathcal{S}_\wedge$ and $\prod$ preserve canonical forms.*

**Proof:** If $\phi_1$ and $\phi_2$ are in canonical form, then $\mathcal{S}_\vee(\phi_1, \phi_2)$ is in canonical form, since *absorb* preserves canonical forms (lemma A.3). By lemma A.4, *product*$(\phi_1, \phi_2)$ is in standard form, and every conjunction in *product*$(\phi_1, \phi_2)$ is in canonical form. By lemma A.3, *absorb*(*product*$(\phi_1, \phi_2)$) is in canonical form. Hence, $\mathcal{S}_\wedge(\phi_1, \phi_2)$ is in canonical form.

$\prod$ preserves canonical forms, since $\prod_\wedge$ preserves canonical forms (lemma A.6) and *absorb* preserves canonical forms (lemma A.3). ■

## A.2 Termination

**Lemma A.8** *Let $\psi = \bigwedge_{i=1}^n (X_i = t_i)$ and $\psi' = \bigwedge_{j=1}^m (X'_j = t'_j)$ be two conjunctions in canonical form. Then, $\forall_j \exists_i\ X_i \equiv X'_j$ whenever $\psi \Rightarrow \psi'$.*

**Proof:** To the contrary, assume that there is a $X'_j$ that is distinct from every $X_i$. Consider some substitution, $\sigma$, that satisfies $\psi$. We have the following two cases, depending on structure of the term $t'_j$.

*Case 1:* $t'_j = f(\ldots)$ for some function symbol $f$. Consider the substitution $\sigma$ obtained by extending $\sigma$ as: $\sigma' = \sigma \cup \{X'_j = f'(\ldots)\}$ where $f'$ is a function symbol distinct from $f$. Clearly, $\sigma'$ satisfies $\psi$, but cannot be extended to satisfy $\psi'$. Hence $\psi \not\Rightarrow \psi'$.

*Case 2:* $t'_j$ is a variable, say $Z$. Since $\psi'$ is in canonical form, $Z$ appears in some other term, say $t'_k$. If $X'_k$ occurs in $\psi$, then let $\sigma(X'_k) = s$; otherwise, let $s$ be any term. Then extend $\sigma$ to $\sigma' = \sigma \cup \{(X'_j = s'), (X'_k = s)\}$ where $s'$ is some term such that $s'$ is not a subterm of $s$. Again, $\sigma'$ satisfies $\psi$ but cannot be extended to satisfy $\psi'$. Hence $\psi \not\Rightarrow \psi'$.

■

**Lemma A.9** *Let $\psi = \bigwedge_{i=1}^n (X_i = t_i)$ and $\psi' = \bigwedge_{j=1}^m (X'_j = t'_j)$ be in standard form. Then, $\forall_j \exists_i\ X'_j \equiv X_i$ and $t'_j$ subsumes $t_i$ whenever $\psi \Rightarrow \psi'$.*

**Proof:** From definition of $\Rightarrow$, every substitution $\sigma$ that satisfies $\psi$ can be extended to satisfy $\psi'$. Hence if $X'_j \equiv X_i$ then every instance of $t_i$ also an instance of $t'_j$ and hence $t'_j$ subsumes $t_i$. Existence of equivalent variables is assured by lemma A.8. ■

For each $t'_j$, the $t_i$ that satisfies the condition in the above lemma is called its corresponding term. The subsumption condition means that at all positions where $t'_j$ has a function symbol, $t_i$ must have the same function symbol. Hence the constant portion of $t'_j$ is contained in the constant portion of $t_i$. We formalize the notion of position as follows:

**Definition A.1 (Position)** *A position is either*

- *the empty string* $\Lambda$ *that reaches the root of the term, or*
- $p.i$*, where* $p$ *is a position and* $i$ *is an integer, which reaches the* $i^{th}$ *child of the root of the subterm reached by* $p$*.*

*The subterm of* $t$ *reached by* $p$ *is denoted by* $t|p$*.*

Now we show that whenever $\psi \Rightarrow \psi'$ and two terms in $\psi'$ share a subterm, the corresponding subterms in $\psi$ must be identical.

**Lemma A.10** *Let* $\psi \Rightarrow \psi'$ *and let* $t'_1$, $t'_2$ *be two terms such that* $(X_1 = t'_1)$ *and* $(X_2 = t'_2)$ *occur in* $\psi'$ *and* $p_1$ *and* $p_2$ *be positions in* $t'_1$ *and* $t'_2$ *respectively, such that* $t'_1|p_1 = t'_2|p_2$*; i.e.,* $t'_1$ *and* $t'_2$ *share a subterm. Then the corresponding terms* $t_1$ *and* $t_2$ *in* $\psi$ *are such that* $t_1|p_1 = t_2|p_2$*.*

**Proof:** Since $t'_1$ subsumes $t_1$ and $t'_2$ subsumes $t_2$, the positions $p_1$ and $p_2$ are defined in $t_1$ and $t_2$ respectively. Let $t'_1|p_1 = t'_2|p_2$, but $t_1|p_1 \neq t_2|p_2$. Then we can find a substitution $\sigma$ that satisfies $\psi$ such that $\sigma(X_1)|p_1 \neq \sigma(X_2)|p_2$. This substitution clearly cannot be extended to satisfy $\psi'$, since any substitution $\sigma'$ that satisfies $\psi'$ must have $\sigma'(X_1)|p_1 = \sigma'(X_2)|p_2$. ■

**Theorem A.11** *Canonical forms are unique (modulo renaming of existential variables) for conjunctions. That is, if* $\psi_1$ *and* $\psi_2$ *are two conjunctions in canonical form and* $\psi_1 \Leftrightarrow \psi_2$*, then* $\psi_1 \equiv \psi_2$ *(modulo renaming of existential variables).*

**Proof:** Follows from lemmas A.8, A.9, and A.10. ■

**Lemma A.12** *Let* $\psi = \bigwedge_{i=1}^{n}(X_i = t_i)$ *and* $\psi' = \bigwedge_{j=1}^{m}(X'_j = t'_j)$ *be two conjunctions in standard form. Then,* $\psi \Rightarrow \psi'$ *whenever*

1. $\forall_j \exists_i \; X'_j \equiv X_i$ *and* $t'_j$ *subsumes* $t_i$*; and*
2. *if* $t'_{i_1}$ *and* $t'_{i_2}$ *are two terms such that* $(X_{i_1} = t'_{i_1})$ *and* $(X_{i_2} = t'_{i_2})$ *occur in* $\psi'$ *with positions* $p_{i_1}$ *and* $p_{i_2}$ *in* $t'_{i_1}$ *and* $t'_{i_2}$ *respectively, such that* $t'_{i_1}|p_{i_1} = t'_{i_2}|p_{i_2}$*; then the corresponding terms* $t_{i_1}$ *and* $t_{i_2}$ *in* $\psi$ *are such that* $t_{i_1}|p_{i_1} = t_{i_2}|p_{i_2}$ *(subterms in* $\psi$ *are shared whenever corresponding subterms in* $\psi'$ *are shared).*

**Proof:** Let $\sigma$ be a substitution satisfying $\psi$, and let $(X'_j = t'_j)$ be an arbitrary constraint in $\psi'$. By condition 1 there exists $(X_i = t_i)$ in $\psi$ such that $X_i \equiv X'_j$ and $t'_j$ subsumes $t_i$. For every variable subterm $Z$ of $t'_j$, let $t$ be the corresponding subterm of $t_i$, and extend $\sigma$ to $\sigma'$ by $\sigma'(Z) = \sigma(t)$. Since $t'_j$ subsumes $t_i$, the corresponding subterm in $t'_j$ of any variable subterm in $t_i$ must be a variable, so the extension of $\sigma$ to $\sigma'$ can always be made. Furthermore, condition 2 ensures that $\sigma'$ is consistent across all terms in $\psi'$. Thus, $\sigma$ can always be extended to $\sigma'$ satisfying $\psi'$. Hence, $\psi \Rightarrow \psi'$. ■

**Lemma A.13** *$approx_k$ is finitely computable and is monotonic with respect to $\succeq$.*

**Proof:** *$approx_k$* clearly terminates when the input conjunction is finite. For monotonicity, we first show that *$approx_k$* is monotonic with respect to $\Rightarrow$. Let $\psi = \bigwedge_{i=1}^{n}(X_i = t_i)$ and $\psi' = \bigwedge_{j=1}^{m}(X'_j = t'_j)$ be two conjunctions in standard form, such that $\psi_1 \Rightarrow \psi_2$. By lemma A.9, $\forall_j \exists_i \;\; X'_j \equiv X_i$ and $t'_j$ subsumes $t_i$; and by lemma A.10, any two terms in $\psi$ have identical subterms whenever the corresponding subterms in $\psi'$ are identical. Since *$approx_k$* replaces identical subterms at depth-$k$ with identical variables, the subsumption and sharing conditions hold for *$approx_k(\psi)$* and *$approx_k(\psi')$*. Thus, by lemma A.12, *$approx_k(\psi) \Rightarrow approx_k(\psi')$*. Since the orders $\Rightarrow$ and $\succeq$ coincide for conjunctions, it follows that *$approx_k$* is monotonic with respect to $\succeq$. ■

**Lemma A.14** *$\prod_\wedge$ is finitely computable and is monotonic with respect to $\succeq$.*

**Proof:** Procedures *project* and *truncate* clearly terminate when their input is finite, and, since *$approx_k$* is finitely computable (lemma A.13), it follows that $\prod_\wedge$ is finitely computable.

Procedure *project* is monotonic with respect to $\succeq$, since, given two sets of constraints corresponding to conjunctions in standard form, *project* removes constraints for the same variables from each set. Since *truncate* removes only constraints equivalent to *True*, it is also monotonic. Thus, since *$approx_k$* is monotonic with respect to $\succeq$ (lemma A.13), it follows that $\prod_\wedge$ is monotonic with respect to $\succeq$. ■

**Lemma A.15** unify *is finitely computable and is monotonic with respect to $\succeq$.*

**Proof:** Let $\psi \Rightarrow \psi'$. Then, *unify*$(\psi, \psi_1) \Leftrightarrow (\psi \wedge \psi_1) \Rightarrow (\psi' \wedge \psi_1) \Leftrightarrow$ *unify*$(\psi', \psi_1)$. Hence, *unify* is monotonic with respect to $\Rightarrow$, and since the orders $\Rightarrow$ and $\succeq$ coincide over conjunctions, is monotonic with respect to $\succeq$.

The unification procedure *mgu* over a finite set of term equations is finitely computable. Procedure *project* clearly terminates when the size of the input set is finite. Therefore, *unify* is finitely computable. ■

**Lemma A.16** *Implication among conjunctions is finitely computable.*

**Proof:** We define implication checking among conjunctions based on the following observation:

$$(\psi_1 \Rightarrow \psi_2) \Leftrightarrow (\mathit{unify}(\psi_1, \psi_2) \Leftrightarrow \psi_1)$$

which follows from the losslessness of *unify* (lemma A.2). Now, since *unify* computes canonical forms, and canonical forms for conjunctions are unique modulo renaming of existential variables, $(unify(\psi_1, \psi_2) \Leftrightarrow \psi_1) \Leftrightarrow (unify(\psi_1, \psi_2) \equiv \psi_1)$. Since *unify* is finitely computable (lemma A.15), this lemma follows. ■

**Lemma A.17** product *is finitely computable and is monotonic with respect to* $\succeq$.

**Proof:** Let $\phi \succeq \phi'$ and $\phi''$ be some conjunction. Then $product(\phi, \phi'')$ consists of conjunctions of the form $unify(\phi_i, \phi''_k)$. Since $\phi \succeq \phi'$, $\exists \psi'_j$ in $\phi'$ such that $\psi_i \succeq \psi'_j$. From monotonicity of *unify* (lemma A.15), $unify(\phi_i, \phi''_k) \succeq unify(\phi'_j, \phi''_k)$, and $unify(\phi'_j, \phi''_k)$ is a conjunction in $product(\phi', \phi'')$. Thus, for every conjunction $\psi$ in $product(\phi, \phi'')$ there is a conjunction $\psi'$ in $product(\phi', \phi'')$ such that $\psi \succeq \psi'$ and hence $product(\phi, \phi'') \succeq product(\phi', \phi'')$. ■

**Lemma A.18** absorb *is finitely computable and is monotonic with respect to* $\succeq$.

**Proof:** Clearly, $absorb(\phi) \succeq \phi$, since *absorb* throws away some conjunctions and never introduces new conjunctions. Let $\phi \succeq \phi'$. Hence, $\forall \psi_i$ in $\phi$, $\exists \psi'_j$ in $\phi'$ such that $\psi_i \succeq \psi' j$. Now we show that $\phi \succeq absorb(\phi')$. If $absorb(\phi') = \phi'$ then $\phi \succeq absorb(\phi')$, since $\phi \succeq \phi'$. If $absorb(\phi') \neq \phi'$, there are two distinct conjunctions in $\phi'$ such that $\psi'_j \Rightarrow \psi'_{j'}$. Hence $absorb(\phi')$ contains only $\psi'_{j'}$. Let $i$ be such that $\psi_i \Rightarrow \psi'_j$. Now, $\psi_i \Rightarrow \psi'_{j'}$ and hence $\phi \succeq absorb(\phi')$, and we get $absorb(\phi) \succeq \phi \succeq absorb(\phi')$. ■

**Theorem A.19** $\mathcal{S}_\wedge$, $\mathcal{S}_\vee$ *and* $\prod$ *are finitely computable and are monotonic with respect to* $\succeq$.

**Proof:** Monotonicity and finite computability of $\mathcal{S}_\wedge$ and $\mathcal{S}_\vee$ follow from monotonicity and finite computability of *absorb* (lemma A.18) and *product* (lemma A.17).

Monotonicity and finite computability of $\prod$ follows from monotonicity and finite computability of *absorb* (lemma A.18) and $\prod_\wedge$ (lemma A.17). ■

**Theorem A.20** *Range of* $\prod$ *is finite.*

**Proof:** If the depth of terms on the rhs of every constraint is bounded, and the number of universally quantified variables is finite, then the number of distinct (modulo renaming of existentially quantified variables) of conjunctions in canonical form is finite. Note that for any $\psi$, $\prod_\wedge(\psi)$ contains only terms of depth $k$ or less. Furthermore, $\prod_\wedge(\psi)$ is in canonical form (lemma A.6). Hence the range of $\prod_\wedge$ is finite. From the definition of canonical forms, if the number of conjunctions is finite, then the number of disjunctions is also finite. Thus, since for all $\phi$, $\prod(\phi)$ is in canonical form, the range of $\prod$ is finite. ■