Computer Science at Kent

Determinacy Inference for Logic Programs

Lunjin Lu and Andy King

Technical Report No. 19-04
October 2004

Abstract This paper presents a determinacy inference analysis for logic programs. The analysis infers determinacy conditions that, if satisfied by a call, ensures that it computes at most one answer and that answer is generated only once. The principal component of the technique is a goal-independent analysis of individual clauses. This derives a condition for a call that ensures only one clause in the matching predicate possesses a successful derivation. Another key component of the analysis is backwards reasoning stage that strengthens these conditions to derive properties on a call that assure determinacy. The analysis has applications in program development, implementation and specialisation.

Published by the Computing Laboratory,
University of Kent, Canterbury, Kent, CT2 7NF, UK

1 Introduction

One issue in logic programming is checking whether a given program and goal are deterministic [3, 5, 8], that is, whether the goal has at most one computed answer (and whether that answer is generated only once). As well as being key to efficient parallel [6, 11] and sequential [5, 8] implementations, knowledge of determinacy is important in program development and O'Keefe [16] writes,

> "you should keep in mind which ... predicates are determinate, and when they are determinate, and you should provide comments for your own code to remind you of when your own code is determinate."

This paper presents a determinacy inference analysis. It synthesises a determinacy condition for each predicate that, if satisfied by a call to the predicate, guarantees that there is at most one computed answer for the call and that the answer is produced only once if ever. Determinacy inference generalises determinacy checking; rather than verify that a particular goal is deterministic it deduces, in a *single* application of the analysis, a class of goals that are deterministic. The analysis has the advantage of being goal-independent, requiring human interaction only to inspect the answers. More exactly, the analysis can be activated by directing the analyser to a source file and pressing a button; in contrast to goal-dependent techniques the analysis does not require a top-level goal or module entry point to be specified by the programmer. As well as being applicable to determinacy checking problems, the analysis opens up intriguing new applications. In program development, determinacy inference can deduce those contexts in which a call can be safely executed under the `once` meta-call, thereby detecting opportunities for tuning space behaviour. Moreover, if the class of deterministic goals inferred by the analysis does not match the expectations of the programmer, then the program is possibly buggy, or at least inefficient. In program specialisation, determinacy inference is promising as a form of binding-time analysis in the so-called semi-online (or mixline) approach to program specialisation [13]. In this scheme, the usual static and dynamic polarisation that is used within classic binding-time analysis [4] 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 a goal-dependent binding-time analysis can verify whether a determinacy condition for a given call is satisfied and therefore whether the call is determinate. These calls can then be marked as semi since determinate goals are prime candidates for unfolding. These semi calls are also annotated with lightweight conditions that select the clause with which to unfold the call. The net result is more aggressive unfolding. To summarise, this paper makes the following contributions:

- it presents a determinacy inference technique which generalises previously proposed determinacy checking techniques. In practical terms, this simplifies the interface between the programmer and the analyser since the programmer only really needs to inspect the results. The analysis also has applications in the burgeoning area of semi-online program specialisation;
- it shows how determinacy inference can be decomposed into the subproblems of (1) deriving a mutual exclusion condition on each call that ensures that only one matching clause has a successful derivation; (2) applying backward reasoning to enrich these conditions on calls so as to assure determinacy;
- it shows that (1) can be tackled with techniques such as depth-k [17] and argument-size analysis [2] when suitably augmented with a projection step; and that (2) can be tackled with backward analysis [12];
- it reports experimental work that demonstrates that the method scales and that it can infer rich (and sometimes surprising) determinacy conditions.

Section 2 illustrates the key ideas with an intuitive worked example. Section 3 explains how mutual exclusion conditions can be derived that, when satisfied by a call, ensures that no more than one clause of the matching predicate can lead to a successful derivation. Section 4 presents a backward analysis that strengthens these conditions to obtain determinacy conditions. Section 5 details an initial experimental evaluation and sections 6 and 7 the related work and conclusions. To make the ideas accessible to a wider programming language audience, the analysis is, wherever possible, presented informally with minimal notation.

2 Worked example

This section explains the analysis by way of an example. In order to derive conditions on calls that are sufficient for determinacy, it is necessary to reason about individual success patterns of the constituent clauses of a predicate. In particular, it is necessary to infer conditions under which the success patterns for any pair of clauses do not overlap. This can be achieved by describing success patterns with suitable abstractions. One such abstraction can be constructed from the list-length norm that is defined as follows:

$$\|t\| = \begin{cases} t & \text{if } t \text{ is a variable} \\ 1 + \|t_2\| & \text{if } t = [t_1|t_2] \\ 0 & \text{otherwise} \end{cases}$$

The norm maps a term to a size that is in fact a linear expression defined over the natural numbers and the variables occurring in the term. Observe that if two terms t_1 and t_2 are described by constant expressions of different value, that is $\|t_1\| \neq \|t_2\|$, then t_1 and t_2 are distinct. In fact, to reason about non-overlapping sets of success patterns (rather than sets of terms), it is necessary to work with argument-size relationships [2] which are induced from a given norm. To illustrate argument-size relationships, and their value in determinacy inference, the following program will be used as a running example throughout this section. The program, like all those consider in the paper, is flat in the sense that the arguments of atoms are vectors of distinct variables. Clauses are number for future reference.

```
(1) append(Xs, Ys, Zs) :-          (3) rev(Xs,Ys) :-
        Xs = [], Ys = Zs.                  Xs = [], Ys = [].
(2) append(Xs, Ys, Zs) :-          (4) rev(Xs,Ys) :-
        Xs = [X|Xs1],                      Xs = [X|Xs1], Ys2 = [X],
        Zs = [X|Zs1],                      rev(Xs1, Ys1),
        append(Xs1, Ys, Zs1).              append(Ys1, Ys2, Ys).
```

2.1 Computing success patterns

To derive size relationships the program is abstracted by applying the norm to the terms occurring within it. Applying the norm to the terms in a syntactic equation $t_1 = t_2$ yields a linear equation $\|t_1\| = \|t_2\|$. The key idea is that a variable in the derived program – the so-called abstract program – describes the size of the corresponding variable in the original program. Since term sizes are non-negative, it is safe to additionally assert that each variable in the abstract program is non-negative. The abstract program thus obtained is listed below.

```
(1) append(Xs, Ys, Zs) :-          (3) rev(Xs, Ys) :-
        Xs ≥ 0,Ys ≥ 0,Zs ≥ 0,              Xs ≥ 0,Ys ≥ 0,
        Xs = 0, Ys = Zs.                   Xs = 0, Ys = 0.
(2) append(Xs, Ys, Zs) :-          (4) rev(Xs,Ys) :-
        Xs ≥ 0,Ys ≥ 0,Zs ≥ 0,              Xs ≥ 0,Ys ≥ 0,
        Xs1 ≥ 0,Zs1 ≥ 0,                   Xs1 ≥ 0,Ys1 ≥ 0,Ys2 ≥ 0,
        Xs = 1 + Xs1,                      Xs = 1 + Xs1, Ys2 = 1,
        Zs = 1 + Zs1,                      rev(Xs1,Ys1),
        append(Xs1, Ys, Zs1).              append(Ys1,Ys2,Ys).
```

The value of the abstract program is that its success patterns, which are given below, describe size attributes of the original program. The key idea is that success sets of the abstract program faithfully describes a size relationships on the success sets of the original program.

$$\begin{aligned} \texttt{append}(x_1, x_2, x_3) &\text{ :- } (x_1 \geq 0) \wedge (x_2 \geq 0) \wedge (x_1 + x_2 = x_3) \\ \texttt{rev}(x_1, x_2) &\text{ :- } (x_1 \geq 0) \wedge (x_1 = x_2) \end{aligned}$$

The relation $x_1 + x_2 = x_3$ captures the property that if the original program is called with a goal $\texttt{append}(x_1, x_2, x_3)$ then any computed answer will satisfy the property that the size of x_1 wrt $\|.\|$, when summed with the size of x_2 will exactly coincide with the size of x_3. Moreover, the success patterns that are systems of linear inequalities can be inferred automatically by mimicking the T_P operator [18] and specifically calculating a least fixpoint (lfp) [2].

2.2 Synthesizing mutual exclusion conditions

Mutual exclusion conditions are synthesized next; one condition for each predicate in the program. Such a condition, if satisfied by a call, guarantees that if one clause of the predicate can lead to a solution then no other clauses can do so. For example, one mutual exclusion condition for `append` is that its first argument is bound to a non-variable term. If the first clause leads to a solution, then its head must unify with the call. Thus the second clause cannot match the call and *vice versa*. Notice, that mutual exclusion is not sufficient for determinacy. For instance, the call `append([W|X],Y,Z)` – which satisfies the above mutual exclusion condition – possesses multiple solutions. Mutual exclusion conditions are synthesised by computing success patterns for individual clauses. This is accomplished by evaluating the body of an abstract clause with the predicate-level success patterns. This yields the following clause-level success patterns:

1	$\texttt{append}(x_1, x_2, x_3) \text{ :- } (x_1 = 0) \wedge (x_2 \geq 0) \wedge (x_2 = x_3)$
2	$\texttt{append}(x_1, x_2, x_3) \text{ :- } (x_1 \geq 1) \wedge (x_2 \geq 0) \wedge (x_1 + x_2 = x_3)$
3	$\texttt{rev}(x_1, x_2) \text{ :- } (x_1 = 0) \wedge (x_2 = 0)$
4	$\texttt{rev}(x_1, x_2) \text{ :- } (x_1 \geq 1) \wedge (x_1 = x_2)$

The next step is to compute a rigidity property that guarantees that at most one of its clauses can yield a computed answer. A term t is rigid wrt to a norm $\|.\|$ if $\|t\|$ is a fixed constant. For example, a term t is rigid wrt list-length if t is *not* an open list. More generally, a Boolean function such as $x_1 \wedge (x_2 \leftrightarrow x_3)$ can express rigidity constraints on the arguments of a call; it states that x_1 is rigid wrt $\|.\|$ and that x_2 if rigid iff x_3 is rigid. Suppose now that $p(\vec{x})$:-c_1 and $p(\vec{x})$:-c_2 are success patterns for two clauses. A rigidity constraint on the arguments of $p(\vec{x})$ that is sufficient for mutual exclusion can be computed by:

$$\mathcal{X}_P(p(\vec{x})) = \bigvee\{\wedge Y \mid Y \subseteq var(\vec{x}) \wedge (\overline{\exists}Y(c_1) \wedge \overline{\exists}Y(c_2) = \mathit{false})\}$$

The projection operator $\overline{\exists}Y.(c)$ maps c onto a weaker linear constraint that ranges over variables in the set Y. For example, $\overline{\exists}\{x_2\}.((x_1 \geq 1) \wedge (x_1 = x_2))$ $= (x_2 \geq 1)$. If $\overline{\exists}Y.(c_1) \wedge \overline{\exists}Y.(c_2)$ is unsatisfiable, then the Boolean formula $\wedge Y$ expresses a rigidity condition on the arguments of $p(\vec{x})$ that is sufficient for mutual exclusion. To see this, observe that if the arguments in Y are rigid, then their sizes cannot change as execution proceeds. Thus the projection $\overline{\exists}Y(c_i)$ holds at the selection of the respective clause since it holds at the end of a successful derivation. Since $\overline{\exists}Y(c_1) \wedge \overline{\exists}Y(c_2)$ is unsatisfiable when Y are rigid, $\wedge Y$ is enough for mutual exclusion. This tactic generates the following conditions for the reverse program which states that the clauses of $\texttt{append}(x_1, x_2, x_3)$ are mutually exclusive if either x_1 is rigid or both x_2 and x_3 are rigid.

$$\mathcal{X}_P(\texttt{append}(x_1, x_2, x_3)) = \vee\{\wedge\{x_1\}, \wedge\{x_2, x_3\}, \wedge\{x_1, x_2, x_3\}\} = x_1 \vee (x_2 \wedge x_3)$$
$$\mathcal{X}_P(\texttt{rev}(x_1, x_2)) = \vee\{\wedge\{x_1\}, \wedge\{x_2\}\} = x_1 \vee x_2$$

2.3 Synthesizing determinacy conditions

The last phase in determinacy inference involves calculating a rigidity constraint for each predicate such that any call satisfying the constraint will yield at most one computed answer. This is achieved with backward analysis [12] which computes a greatest fixpoint (gfp) to strengthen the mutual exclusion conditions into determinacy conditions. The gfp makes use of rigidity success patterns which are computed, again by simulating the T_P operator, in a lfp calculation.

2.3.1 Least fixpoint

The lfp calculation is performed to another abstract version of the program obtained, this time, by replacing syntactic constraints with rigidity constraints. For example, `Xs = 1 + Xs1` is replaced with the Boolean formula `Xs` $\leftrightarrow$ `Xs1` which expresses that `Xs` is rigid wrt list-length iff `Xs1` is rigid. The abstract program obtained in this fashion is given below.

```
(1) append(Xs, Ys, Zs) :-            (3) rev(Xs, Ys) :-
        Xs ∧ (Ys ↔ Zs).                      Xs, Ys.
(2) append(Xs, Ys, Zs) :-            (4) rev(Xs,Ys) :-
        Xs ↔ Xs1,                            (Xs ↔ Xs1), Ys2,
        Zs ↔ Zs1,                            rev(Xs1, Ys1),
        append(Xs1, Ys, Zs1).                append(Ys1, Ys2, Ys).
```

Interpreting this program with a version of T_P that operates over Boolean functions, the following success patterns are obtained that express rigidity properties of the original program. The pattern for `rev`, for instance, states x_1 is rigid iff x_2 is rigid in any computed answer to $\texttt{rev}(x_1, x_2)$ in the original program.

$$\texttt{append}(x_1, x_2, x_3) \text{ :- } x_1 \wedge (x_2 \leftrightarrow x_3) \qquad \texttt{rev}(x_1, x_2) \text{ :- } x_1 \leftrightarrow x_2$$

2.3.2 Greatest fixpoint

Each iteration in the gfp calculation amounts to:

- deriving a determinacy condition for each clause of the predicate that ensures no more than one derivation commencing with that clause may succeed;
- conjoining these conditions with the mutual exclusion of the predicate.

The single resulting condition, which is expressed as rigidity constraint, defines the next iterate. The iterates that arise when processing reverse are given below:

$$I_0 = \left\{ \begin{array}{rl} \texttt{append}(x_1, x_2, x_2) & \text{:- } \mathit{true} \\ \texttt{rev}(x_1, x_2) & \text{:- } \mathit{true} \end{array} \right\}$$

$$I_1 = \left\{ \begin{array}{rl} \texttt{append}(x_1, x_2, x_2) & \text{:-}\ x_1 \vee (x_2 \wedge x_3) \\ \texttt{rev}(x_1, x_2) & \text{:-}\ x_1 \vee x_2 \end{array} \right\}$$

$$I_3 = I_2 = \left\{ \begin{array}{rl} \texttt{append}(x_1, x_2, x_2) & \text{:-}\ x_1 \vee (x_2 \wedge x_3) \\ \texttt{rev}(x_1, x_2) & \text{:-}\ x_1 \end{array} \right\}$$

To illustrate the gfp consider computing the determinacy condition for `rev` in I_2. The first clause of `rev` possesses body atoms which are deterministic builtins. Thus the rigidity condition of *true* is trivially sufficient for this clause to be deterministic. Consider now the second clause. The two calls in its body, `rev(Xs1,Ys1)` and `append(Ys1,Ys2,Ys)`, are processed separately as follows:

- The determinacy condition in I_1 for `rev(Xs1,Ys1)` is $\texttt{Xs1} \vee \texttt{Ys1}$ and the combined success pattern of the equations that precede it is $(\texttt{Xs} \leftrightarrow \texttt{Xs1}) \wedge \texttt{Ys1}$. Thus `rev(Xs1,Ys1)` is deterministic if $((\texttt{Xs} \leftrightarrow \texttt{Xs1}) \wedge \texttt{Ys1}) \rightarrow (\texttt{Xs1} \vee \texttt{Ys1})$ holds upon entry to the body of the clause.

- The determinacy condition in I_1 for `append(Ys1,Ys2,Ys)` is $\texttt{Ys1} \vee (\texttt{Ys2} \wedge \texttt{Ys})$. The combined success pattern of the equations and the call `rev(Xs1,Ys1)` that precede it is $(\texttt{Xs} \leftrightarrow \texttt{Xs1}) \wedge \texttt{Ys2} \wedge (\texttt{Xs1} \wedge \texttt{Ys1}) = \texttt{Xs} \wedge \texttt{Xs1} \wedge \texttt{Ys1} \wedge \texttt{Ys2}$. The call `append(Ys1,Ys2,Ys)` is deterministic if $(\texttt{Xs} \wedge \texttt{Xs1} \wedge \texttt{Ys1} \wedge \texttt{Ys2}) \rightarrow (\texttt{Ys1} \vee (\texttt{Ys2} \wedge \texttt{Ys}))$ holds when the body is entered.

These conditions for determinacy of `rev(Xs1,Ys1)` and `append(Ys1,Ys2,Ys)` are then conjoined to give, say f, (in this case conjunction is trivial since the condition for `append` is *true*). The condition f is formulated in terms of the rigidity of variables occurring in the body of the `rev` clause. What is actually required is a condition on a `rev` call that is sufficient for determinacy. Thus those variables in f that do not occur in the clause head, namely `Xs1` and `Ys2`, are eliminated from f to obtain a condition sufficient for the call `rev(Xs,Ys)` to be deterministic. Eliminating `Xs1` and `Ys2` from f (in the manner prescribed in section 4) gives `Xs`. If `Xs` holds, then f holds. Hence if the second clause is selected and the call possess a rigid first argument, then determinacy follows.

Finally, determinacy conditions for the two `rev` clauses are conjoined with the mutual exclusion condition to obtain $\mathit{true} \wedge \texttt{Xs} \wedge (\texttt{Xs} \vee \texttt{Ys}) = \texttt{Xs}$. Thus the call `rev(Xs,Ys)` is guaranteed to be deterministic if `Xs` is rigid.

3 Synthesising mutual exclusion conditions

To compute mutual exclusion conditions, it is necessary to characterise successful computations at level of clauses. Specifically, it is necessary to characterise the set of solutions for any call that can be obtained with a derivation that commences with a given clause. Success pattern analysis can be adapted to this task.

3.1 Success patterns

Example 3.1 *To illustrate a success pattern analysis other than argument-size analysis, consider a depth-k analysis of the Quicksort program listed below.*

```
(1) sort(X,Y) :- L = [], qsort(X,Y,L).

(2) qsort(X,S,T) :- X = [], S = T.
(3) qsort(X,S,T) :- X = [Y|X1], M1 = [Y|M],
        part(X1,Y,L,G), qsort(L,S,M1), qsort(G,M,T).

(4) part(X,M,L,G) :- Xs = [], L = [], G = [].
(5) part(X,M,L,G) :-
        X = [Y|X1], L = [Y|L1], Y ≤ M, part(X1,M,L1,G).
(6) part(X,M,L,G) :-
        X = [Y|X1], G = [Y|G1], Y > M, part(X1,M,L,G1).
```

In this context of depth-k analysis, a success pattern is an atom paired with a Herbrand constraint were the terms occurring in the constraint have a depth that does not exceed a finite bound k. The success patterns for the predicates and clauses are given below, to the left and the right, respectively.

$$\begin{array}{r}
\texttt{sort}(x_1, x_2) \text{ :- } \mathit{true} \\
\texttt{qsort}(x_1, x_2, x_3) \text{ :- } \mathit{true} \\
\texttt{part}(x_1, x_2, x_3, x_4) \text{ :- } \mathit{true}
\end{array}
\quad
\begin{array}{lrl}
1 & \texttt{sort}(x_1, x_2) & \text{:- } \mathit{true} \\
2 & \texttt{qsort}(x_1, x_2, x_3) & \text{:- } x_1 = [], x_2 = x_3 \\
3 & \texttt{qsort}(x_1, x_2, x_3) & \text{:- } x_1 = [_|_], x_2 = [_|_] \\
4 & \texttt{part}(x_1, x_2, x_3, x_4) & \text{:- } x_1 = [], x_3 = [], x_4 = [] \\
5 & \texttt{part}(x_1, x_2, x_3, x_4) & \text{:- } x_1 = [y|_], x_3 = [y|_], y \leq x_2 \\
6 & \texttt{part}(x_1, x_2, x_3, x_4) & \text{:- } x_1 = [y|_], x_4 = [y|_], y > x_2
\end{array}$$

3.2 Mutual exclusion conditions

The technique previously introduced for synthesising mutual exclusion conditions is formulated in terms of argument-size analysis and rigidity analysis and the relationship between rigidity and size; rigidity constraints are used to specify conditions under which pairs of size abstractions are incompatible. To generalise these ideas to other domains, such as depth-k, it is necessary to generalise the concept of norm and replace it with a mapping ν from the set of terms to a set of abstract terms. The concept of rigidity is still meaningful in this general setting: a term t is rigid wrt ν iff $\nu(\theta(t)) = \nu(t)$ for every substitution $\theta \in \mathrm{Sub}$. Let $rigid_\nu$ be the rigidity predicate on terms induced by ν, that is, $rigid_\nu(t)$ holds if t is rigid wrt ν. For example, if ν is depth-2 abstraction [17] then $rigid_\nu(t)$ holds iff all the variables in t occur at depth 2 or more.

Mutual exclusion conditions are expressed within a dependency domain that can specify rigidity requirements. The property that all the variables in a term occur at or beneath level k can be tracked within the dependency domain, but it is simpler to trace a property that implies rigidity (rather than the induced rigidity property itself). Hence let $rigid'_\nu$ denote any predicate such that $rigid_\nu(t)$ holds if $rigid'_\nu(t)$ holds. For instance, $rigid'_\nu(t) = ground(t)$. Such a predicate can then induce abstraction $\alpha_{\mathrm{rigid}'_\nu} : \wp(\mathrm{Sub}) \rightarrow \mathrm{Pos}$ and concretisation $\gamma_{\mathrm{rigid}'_\nu} : \mathrm{Pos} \rightarrow \wp(\mathrm{Sub})$ maps between Pos and sets of substitutions as follows:

$$\begin{aligned} \gamma_{\mathrm{rigid}'_\nu}(f) &= \{\theta \in \mathrm{Sub} \mid \forall \kappa \in Sub.\mathrm{assign}(\kappa \circ \theta) \models f\} \\ \alpha_{\mathrm{rigid}'_\nu}(\Theta) &= \wedge\{f \in \mathrm{Pos} \mid \Theta \subseteq \gamma_{\mathrm{rigid}'_\nu}(f)\} \end{aligned}$$

where $\mathrm{assign}(\theta) = \wedge\{x \leftrightarrow \mathrm{rigid}'_\nu(\theta(x)) \mid x \in \mathrm{dom}(\theta)\}$. Note that although the underlying domain is Pos, the underlying abstraction is not necessarily groundness. Indeed, the predicate rigid'_ν parameterises the meaning of $\alpha_{\mathrm{rigid}'_\nu}$ and $\gamma_{\mathrm{rigid}'_\nu}$ and these maps define the interpretation of Pos.

Now that a target domain exists in which determinacy conditions can be expressed, it remains to define a general procedure for inferring these conditions. Let $p(\vec{x})$:-c_1 and $p(\vec{x})$:-c_2 be the success patterns of two clauses C_1 and C_2 of $p(\vec{x})$ where c_1 and c_1 are abstract term constraints such as depth-k abstractions. The following predicate checks if the condition $\wedge_{y \in Y}\mathrm{rigid}'_\nu(y)$ is enough for C_1 and C_2 to be mutually exclusive on $p(\vec{x})$.

$$\mathcal{X}_P(Y, p(\vec{x}), C_1, C_2) = (\overline{\exists}Y(c_1) \wedge \overline{\exists}Y(c_2) = false)$$

The following proposition (whose proof is given in [14]) formalises the intuitive argument that was given in section 2.2.

Proposition 3.1 Let $\theta \in \mathrm{Sub}$ and $Y \subseteq var(\vec{x})$. Suppose $\mathcal{X}_P(Y, p(\vec{x}), C_1, C_2)$ holds and $\wedge_{y \in Y} rigid'_\nu(\theta(y))$ holds. Then

- all derivations of $\theta(p(\vec{x}))$ using C_1 as the first clause fail or
- all derivations of $\theta(p(\vec{x}))$ using C_2 as the first clause fail.

Now let S denote the set of clauses that define $p(\vec{x})$. A rigidity constraint $\wedge Y$ is a mutual exclusion condition for $p(\vec{x})$ iff it is a mutual exclusion condition for all pairs of clauses drawn from S. This observation leads to the following:

$$\mathcal{X}_P(p(\vec{x})) = \bigvee \{\wedge Y \mid \forall C_1, C_2 \in S.(C_1 \neq C_2 \rightarrow \mathcal{X}_P(Y, p(\vec{x}), C_1, C_2)\}$$

The following corollary of proposition 3.1 verifies that $\mathcal{X}_P(p(\vec{x}))$ is truly a mutual exclusion condition for $p(x)$.

Corollary 3.1 If $\alpha_{\mathrm{rigid}'_\nu}(\{\theta\}) \models \mathcal{X}_P(p(\vec{x}))$ then at most one clause of $p(\vec{x})$ can lead to a successful derivation of $\theta(p(\vec{x}))$.

Example 3.2 *The left-hand column gives the exclusion conditions for the quicksort program, synthesised from the clause-level depth-k abstractions listed in example 3.1 and using the predicate $rigid'_\nu(t) = ground(t)$. The same predicate was used to generate the exclusion conditions in the right-hand column using argument-size abstractions (not provided).*

$$\begin{array}{rlrl} \mathcal{X}_P(\texttt{sort}(x_1, x_2)) = & true & \mathcal{X}_P(\texttt{sort}(x_1, x_2)) = & true \\ \mathcal{X}_P(\texttt{qsort}(x_1, x_2, x_3)) = & x_1 & \mathcal{X}_P(\texttt{qsort}(x_1, x_2, x_3)) = & x_1 \vee (x_2 \leftrightarrow x_3) \\ \mathcal{X}_P(\texttt{part}(x_1, x_2, x_3, x_4)) = & x_1 \wedge x_2 & \mathcal{X}_P(\texttt{part}(x_1, x_2, x_3, x_4)) = & false \end{array}$$

Note that weaker requirements for mutual exclusion can be obtained by combining these two sets of conditions. Note too that these conditions can only be combined by operating in a domain defined in terms of a common predicate $ground(t)$ which is stronger than both $rigid_{\|.\|}(t)$ and $rigid_{\mathrm{depth}-k}(t)$.

4 Synthesising determinacy conditions

This section revisits the backward analysis that strengthens mutual exclusion conditions to obtain the determinacy conditions. As with the previous section, the presentation focusses on those left open in the worked example section.

4.1 Abstracting the program for general rigidity

The exercise of specifying $\alpha_{\text{rigid}'_\nu}$ and $\gamma_{\text{rigid}'_\nu}$ is more than an aesthetic predilection. It provides a mechanism for deriving an abstract program that captures rigidity relationships between program variables where the notion of rigidity is specified by rigid'_ν. Consider first how to abstract an equation $t_1 = t_2$ in the context of rigid'_ν. The relationship between an equation and its most general unifiers (mgus) is such that the equation can be safely described by any Boolean function f such that $\alpha_{\text{rigid}'_\nu}(\{\theta\}) \models f$ where θ is any mgu of $t_1 = t_2$. For example, if $\nu'(t) = \|t\|$ then $x_1 \leftrightarrow x_3$ describes $x_1 = [x_2|x_3]$. To see this, let $\kappa \in \text{Sub}$ and observe that $\theta = \{x_1 \mapsto [x_2|x_3]\}$ is a mgu of the equation $x_1 = [x_2|x_3]$. Then

$$\text{rigid}'_\nu(\kappa \circ \theta(x_3)) = \text{rigid}'_\nu(\kappa(x_3)) = \text{rigid}'_\nu([\kappa(x_2)|\kappa(x_3)]) = \text{rigid}'_\nu(\kappa \circ \theta(x_1))$$

Thus $\text{assign}(\kappa \circ \theta) \models (x_1 \leftrightarrow x_3)$ for all $\kappa \in \text{Sub}$, whence it follows that $x_1 \leftrightarrow x_3$ describes $x_1 = [x_2|x_3]$. If $\text{rigid}'_\nu(t) = \mathit{ground}(t)$ then $t_1 = t_2$ is abstracted by $\wedge\{x \leftrightarrow \wedge \mathit{vars}(\theta(x)) \mid x \in \mathit{dom}(\theta)\}$ where θ is a mgu of the equation $t_1 = t_2$, though $(\wedge \mathit{vars}(t_1)) \leftrightarrow (\wedge \mathit{vars}(t_2))$ is a simpler, albeit less precise, abstraction. A call to a builtin $p(\vec{x})$ can be handled by abstracting it with any function f such that $\alpha_{\text{rigid}'_\nu}(\Theta) \models f$ where Θ is the set of computed answers for $p(\vec{x})$. For instance, if $\nu'(t) = \mathit{ground}(t)$ then $x_1 \wedge x_2$ describes the call (x_1 `is` x_2) whereas if $\nu'(t) = \|t\|$ then $x_1 \wedge x_2$ describes the builtin `length`(x_1, x_2).

Example 4.1 *The following rigidity program is obtained from the quicksort program using* $\text{rigid}'_\nu(t) = \mathit{ground}(t)$.

```
(1) sort(X, Y) :- L, qsort(X, Y, L).
(2) qsort(X, S, T) :- X, S ↔ T.
(3) qsort(X, S, T) :- X ↔ (Y ∧ X1), M1 ↔ (Y ∧ M),
        part(X1, Y, L, G), qsort(L, S, M1), qsort(G, M, T).

(4) part(X, M, L, G) :- Xs, L, G.
(5) part(X, M, L, G) :-
        X ↔ (Y ∧ X1), L ↔ (Y ∧ L1), Y, M, part(X1,M,L1,G).
(6) part(X, M, L, G) :-
        X ↔ (Y ∧ X1), G ↔ (Y ∧ G1), Y, M, part(X1,M,L,G1).
```

Example 4.2 *Once the abstract program is defined, the rigidity success patterns can be calculated in the manner previously described to give:*

$$\begin{aligned}
\mathtt{part}(x_1, x_2, x_3, x_4) &\text{ :- } x_1 \wedge x_3 \wedge x_4 \\
\mathtt{qsort}(x_1, x_2, x_3) &\text{ :- } x_1 \wedge (x_2 \leftrightarrow x_3) \\
\mathtt{sort}(x_1, x_2) &\text{ :- } x_1 \leftrightarrow x_2
\end{aligned}$$

4.2 Determinacy conditions

Synthesis of determinacy conditions commences by assuming that all calls are trivially determinate, that is, the condition *true* is sufficient for determinacy. These conditions are then checked by reasoning backwards across all clauses. If a condition turns out to be too weak then it is strengthened and the whole process is repeated until the conditions are verified to be sufficient for determinacy. One of the more subtle aspects of this procedure relates to variable elimination. If a condition f, defined in terms of a variable x is sufficient for determinacy, then it can become necessary to calculate another condition, g say, independent of x which is also sufficient for determinacy. Universal quantification operator $\forall_x : \text{Pos} \mapsto \text{Pos}$ provides a mechanism for doing this:

$$\forall_x(f) = \text{if } f' \in \text{Pos then } f' \text{ else } \mathit{false} \quad \text{where} \quad f' = f[x \mapsto \mathit{true}] \wedge f[x \mapsto \mathit{false}]$$

The significance of this operator is that $\forall_x(f) \models f$, hence if f is sufficient for determinacy, then so is $\forall_x(f)$. To succinctly define the gfp operator it is convenient to define a project onto (rather than project out) operator $\overline{\forall} Y(f) = \forall_{y_1}(\forall_{y_2}(\cdots \forall_{y_n}(f)\cdots))$ where each y_i is a (free) variable occurring in f which does not appear in the set of variables Y; in effect f is projected onto Y.

Example 4.3 *Consider* $\overline{\forall}\{\mathtt{X}, \mathtt{S}, \mathtt{T}\}(e)$ *with* $e = \mathtt{X} \vee (\mathtt{X1} \wedge \mathtt{M1}) \vee (\mathtt{X1} \wedge \mathtt{Y})$. *Now*

$$e[\mathtt{M1} \mapsto \mathit{true}] \wedge e[\mathtt{M1} \mapsto \mathit{false}] = (\mathtt{X} \vee \mathtt{X1}) \wedge (\mathtt{X} \vee (\mathtt{X1} \wedge \mathtt{Y})) = \mathtt{X} \vee (\mathtt{X1} \wedge \mathtt{Y})$$

Thus put $e' = \forall_{\mathtt{M1}}(e) = \mathtt{X} \vee (\mathtt{X1} \wedge \mathtt{Y})$ *and repeating this tactic:*

$$e'[\mathtt{X1} \mapsto \mathit{true}] \wedge e'[\mathtt{X1} \mapsto \mathit{false}] = (\mathtt{X} \vee \mathtt{Y}) \wedge (\mathtt{X}) = \mathtt{X}$$

Hence put $e'' = \forall_{\mathtt{X1}}(e') = \mathtt{X}$ *and it follows* $\overline{\forall}\{\mathtt{X}, \mathtt{S}, \mathtt{T}\}(e) = \mathtt{X}$. *Observe that* $\mathtt{X} \models e$.

The gfp operates on an abstract program P obtained via rigidity abstraction. To express the operator intuitively, the success set of rigidity patterns, denoted S, is considered to be a map from atoms to Pos formulae. Similarly, the rigidity conditions inferred in the gfp, denoted I, are represented as a map from atoms to formulae. The mechanism that the gfp operator uses to successively update I is to replace each pattern $p(\vec{x})$:- $f \in I$ with another $p(\vec{x})$:- $\mathcal{X}_P(p(\vec{x})) \wedge (\wedge F)$ until stability is achieved where the set of Boolean formula F is defined by:

$$F = \left\{ \overline{\forall}\vec{x}(e) \;\middle|\; \begin{array}{c} p(\vec{x}) \,\text{:-}\, f_1, \cdots, f_m, p_1(\vec{x}_1), \cdots, p_n(\vec{x}_n) \in P \\ g_i = (\wedge_{1 \leq k \leq m} f_k) \wedge (\wedge_{1 \leq j < i} S(p(\vec{x}_j))) \\ e = \bigwedge_{1 \leq i \leq n}(g_i \rightarrow I(p_i(\vec{x}_i))) \end{array} \right\}$$

The function $\mathcal{X}_P(p(\vec{x})) \wedge (\wedge F)$ is at least as strong as the formula f it replaces and thus the operator generates a downward iteration sequence. If I' is the gfp thus obtained, the following theorem states how it characterises determinacy.

Theorem 4.1 *If $\theta \in \mathrm{Sub}$ and $\alpha_{\mathrm{rigid}'_\nu}(\{\theta\}) \models I'(p(\vec{x}))$ then $\theta(p(\vec{x}))$ has at most one computed answer.*

Example 4.4 *The iterates that arise when processing the quicksort program are*

$$I_0 = \left\{ \begin{array}{rl} \mathtt{part}(x_1, x_2, x_3, x_4) & \text{:- } \mathit{true}, \\ \mathtt{qsort}(x_1, x_2, x_3) & \text{:- } \mathit{true}, \\ \mathtt{sort}(x_1, x_2) & \text{:- } \mathit{true} \end{array} \right\}$$

$$I_1 = \left\{ \begin{array}{rl} \mathtt{part}(x_1, x_2, x_3, x_4) & \text{:- } x_1 \wedge x_2, \\ \mathtt{qsort}(x_1, x_2, x_3) & \text{:- } x_1, \\ \mathtt{sort}(x_1, x_2) & \text{:- } \mathit{true} \end{array} \right\}$$

$$I_3 = I_2 = \left\{ \begin{array}{rl} \mathtt{part}(x_1, x_2, x_3, x_4) & \text{:- } x_1 \wedge x_2, \\ \mathtt{qsort}(x_1, x_2, x_3) & \text{:- } x_1, \\ \mathtt{sort}(x_1, x_2) & \text{:- } x_1 \end{array} \right\}$$

To illustrate how computation proceeds, consider computing the determinacy condition for `qsort` *in I_2. The first abstract clause for* $\mathtt{qsort(X, S, T)}$ *does not contain any call; hence its determinacy condition is computed as true. The second abstract clause for* $\mathtt{qsort(X, S, T)}$ *has three calls. The first call* $\mathtt{part(X1, Y, L, G)}$ *has a determinacy condition* $\mathtt{X1} \wedge \mathtt{Y}$ *in I_1. The cumulative* *success patterns of the builtins that precede it are* $(\mathtt{X} \leftrightarrow (\mathtt{Y} \wedge \mathtt{X1})) \wedge (\mathtt{M1} \leftrightarrow (\mathtt{Y} \wedge \mathtt{M}))$*. Thus if*

$$e_1 = ((\mathtt{X} \leftrightarrow \mathtt{Y} \wedge \mathtt{X1}) \wedge (\mathtt{M1} \leftrightarrow \mathtt{Y} \wedge \mathtt{M})) \rightarrow (\mathtt{X1} \wedge \mathtt{Y}) = \mathtt{X} \vee (\mathtt{X1} \wedge \mathtt{M1}) \vee (\mathtt{X1} \wedge \mathtt{Y})$$

holds when the body is entered, then $\mathtt{part(X1, Y, L, G)}$ *will be deterministic.*

The second call $\mathtt{qsort(L, S, M1)}$ *has a determinacy condition* $\mathtt{L}$ *in* I_1 *and the success pattern of* $\mathtt{part(X1, Y, L, G)}$ *is* $\mathtt{X1} \wedge \mathtt{L} \wedge \mathtt{G}$*. Moreover the Boolean function* $f = (\mathtt{X} \leftrightarrow \mathtt{Y}) \wedge (\mathtt{M1} \leftrightarrow (\mathtt{Y} \wedge \mathtt{M})) \wedge \mathtt{X1} \wedge \mathtt{L} \wedge \mathtt{G}$ *describes effect of the calls that precede* $\mathtt{qsort(L, S, M1)}$*. Hence* $e_2 = f \rightarrow L = true$ *is a condition which, if it holds at entry to the body, is sufficient for the call* $\mathtt{qsort(L, S, M1)}$ *to be deterministic. Likewise* $e_3 = true$ *is sufficient for the* $\mathtt{qsort(G, M, T)}$ *call to be deterministic. The combined determinacy condition is thus* $e_1 \wedge e_2 \wedge e_3 = e_1$ *and eliminating the body variables which do not occur in the head (using the result from example 4.3) yields* $\overline{\forall}\{\mathtt{X}, \mathtt{S}, \mathtt{T}\}(e_1) = \mathtt{X}$*. Combining this with the mutual exclusion condition gives* $\mathtt{X}$*, thus the determinacy requirement for* `qsort` *does not change.*

An astute reader will have noticed in the worked example section that a call to the $\mathtt{rev}(x_1, x_2)$ predicate is determinate if either x_1 *or* x_2 are rigid. Yet the analysis only infers that the rigidity of x_1 is sufficient for determinacy. If the `rev` and `append` calls in the body of the second `rev` clause are interchanged, however, then the analysis will infer that the rigidity of x_2 is sufficient for determinacy. This would suggest the following revision of the above operator: infer a determinacy condition for each permutation of the body atoms; then apply disjunction to merge these conditions to find a more general condition sufficient for determinate behaviour of that clause. However, this tactic, as well as being potentially inefficient, is also in general wrong. To see this, suppose that the sufficient condition for determinacy for one goal ordering is x_1 and the condition for another is $x_1 \rightarrow x_2$. However, within Pos, $x_1 \vee (x_1 \rightarrow x_2) = true$ and yet *true* is the vacuous condition which places no constraint on the call $\mathtt{rev}(x_1, x_2)$.

5 Experimental evaluation

To evaluate inference analysis, a prototype analyzer has been constructed in SICStus Prolog 3.8.3. The implementation follows sections 3 and 4 closely. The depth-k and argument-size analyses (which applies term-size abstraction) compute success patterns for each clause in the input program, synthe-ses groundness abstractions sufficient for mutual exclusion. These modules

		conditions	
file	*predicate*	*exclusion*	*determinacy*
treesort	`tree_to_list_aux`(x_1, x_2, x_3)	x_1	x_1
	`tree_to_list`(x_1, x_2)	*true*	x_1
	`list_to_tree`(x_1, x_2)	*true*	x_1
	`insert_list`(x_1, x_2, x_3)	x_1	$x_1 \wedge x_2$
	`insert`(x_1, x_2, x_3)	$x_1 \wedge (x_2 \vee x_3)$	$x_1 \wedge (x_2 \vee x_3)$
	`treesort`(x_1, x_2)	*true*	x_1
queens	`noattack`(x_1, x_2, x_3)	x_2	x_2
	`safe`(x_1)	x_1	x_1
	`delete`(x_1, x_2, x_3)	*false*	*false*
	`perm`(x_1, x_2)	$x_1 \vee x_2$	*false*
	`queens`(x_1, x_2)	*true*	*false*
permsort	`select`(x_1, x_2, x_3)	*false*	*false*
	`ordered`(x_1)	x_1	x_1
	`perm`(x_1, x_2)	$x_1 \vee x_2$	*false*
	`sort`(x_1, x_2)	*true*	*false*
serialize	`arrange0`(x_1, x_2)	$x_1 \vee x_2$	x_1
	`numbered`$(x_1,\ x_2,\ x_3)$	x_1	x_1
	`palin`(x_1)	*true*	*true*
	`pairlists`$(x_1,\ x_2,\ x_3)$	$x_1 \vee x_2 \vee x_3$	$x_1 \vee x_2 \vee x_3$
	`serialize0`(x_1, x_2)	*true*	$x_1 \wedge x_2$
	`split0`$(x_1,\ x_2,\ x_3,\ x_4)$	$x_1 \wedge x_2$	$x_1 \wedge x_2$
	`go`(x_1)	*true*	*false*

Figure 1: Precision results for determinacy inference

also produce the abstract program on which subsequent analyses are based. The backward analysis is engineered using much of the machinery described in [12]. One notable difference is that some builtins require special handling; most builtins are determinate but others are determinate only when certain conditions are satisfied. For instance, `current_op`(x_1,x_2,x_3) is determinate only when x_1 is ground.

The analyser has been applied to 50 programs ranging in size between 10 and 4000 LOC. These programs can be found at `http://www.oakland.edu/~l2lu/Benchmarks-Det.zip`. Quantitative precision measures are difficult for determinacy inference because the number of predicates that the programmer intended to be determinate is, in general, unknown. To demonstrate the precision of the analysis, some illustrative results are therefore given for several familiar programs. The results for the first program listed

	argument-size			*depth-k*		
file	*succ*	*lfp*	*gfp*	*succ*	*lfp*	*gfp*
boyer	1666	591	60	441	360	50
bryant	7522	261	90	371	321	80
chat_80	67393	2153	431	494578	4977	631
ga	2146	161	40	2814	290	40
ili	2314	450	111	1222	531	100
ime	653	140	40	120	161	40
nand	17921	1682	421	841	801	260
nbody	877	191	30	241	301	80
peep	404	170	30	761	441	70
peval	6938	621	100	4466	611	90
press	584	251	40	320	381	70
reducer	7867	270	50	190	301	50
rubik	30150	420	70	571	3755	221
sim	10270	561	171	35411	611	100
sim_v5-2	2948	581	170	491	701	180
trs	13174	280	91	321	450	100

Figure 2: Timing results for determinacy inference

in Figure 1 illustrate that the determinacy conditions are often disjunctive reflecting the multi-mode nature of predicates. The second program demonstrates that the analysis will infer *false* for a predicate that is genuinely non-determinate. The third program shows that even the exclusion conditions are themselves interesting. For example, one might have thought that the predicate $\texttt{select}(x_1, x_2, x_3)$ – which selects an element x_1 of the list x_2 to give the residual list x_3 – is determinate if called with x_1, x_2 and x_3 ground. However, the call `select(1, [1,1,2], [1,2])` succeeds twice and as a consequence `sort([1,1,2], L)` manifests the buggy behaviour that it generates the answer `L = [1,1,2]` twice. Finally, the fourth program illustrates a so-called false positive. The top-level predicate $\texttt{go}(x_1)$ appears to be determinate and we conjecture that this can be inferred by replacing the groundness analysis used in the above experiments with a rigidity analysis [10] that is sensitive to the particular structure of the trees that arise in serialize.

To assess scalability, timing experiments were performed on the analysis components using a 2.49GHz PC with 240 MB RAM running XP. Only the timings for the larger programs are given in Figure 2. The *succ*, *lfp* and *gfp* columns give the time in milliseconds required for the argument-size (or depth-k) analysis and calculating the lfp and gfp using the exclusion conditions synthesised from this analysis. Interestingly, the gfp is uniformly faster then the lfp despite the fact that gfp operator is more complicated than the lfp operator. The table shows that the analysis time is dominated by the analyses that feed the client analysis – the backward analysis. There is no reason why the argument size analysis cannot be very significantly improved by replacing a constraint based implementation [2] with one based on a polyhedral library [15]. Moreover, the depth-k analysis would benefit from a more intelligent iteration strategy. Nevertheless, the results demonstrate that determinacy inference is practical even when component analyses are implemented naively.

6 Related work

Giacobazzi and Ricci [11] recognize the value of goal-independence in determinacy analysis and present a solution that tracks deterministic ground dependencies. A ground dependence from a set of input arguments to a set of output arguments is deterministic if, whenever the input arguments are ground, the execution of the predicate binds any output argument to a single ground term. Hence their proposal cannot reason about predicates that compute the same output multiply. The work predates the domains Def and Pos [1] and thus is formulated in terms of hypergraphs. However, even defining an order on hypergraphs is surprising subtle. For example, the ordering on abstract atoms proposed in [11] asserts that $p(g, g)$ is less than the atom $p(ng, ng)$ paired with a deterministic ground dependence from the first argument to the second. Observe, however, that the first abstract atom describes a set of concrete atoms that includes $p(a, b)$ and $p(a, c)$ but the set of concrete atoms described by the second cannot include both. Nevertheless, the proposal is not fundamentally flawed and in our opinion the work is in many ways ahead of its time.

Dawson *et al.* [5] propose an analysis that detects determinacy. The analysis derives a necessary condition for each clause to succeed for a given class of goals. If the necessary conditions for a predicate are pair-wise inconsistent then a call to the predicate is recognized as determinate. The authors do not address the issue of synthesizing sufficient conditions for mutual exclusion.

Determinacy inference was inspired by the modular construction of termination inference [9] which is itself composed of components that include argument-size analysis [2] and backward analysis [12]. In termination inference, size relations are used to deduce grounding conditions sufficient to observe size decreases, and thus termination, on successive recursive calls. Backward analysis is applied to derive sufficient conditions for termination for a compound goal that is executed left-to-right. In determinacy inference, the size issue does not relate primarily to calls but to the relative sizes of the answers generated from different clauses. Determinacy inference also differs from termination inference in that the latter applies the framework of [12] directly whereas the former does not. While the gfp in [12] propagates requirements from right-to-left across the body of a clause; the gfp in this paper propagates determinacy requirements on each call in the body using the conjunction of the success patterns of those calls that precede it. Observe that this conjunction can be pre-computed; it does not need to be reevaluated on each application of gfp operator. Thus, the structure of the gfp presented in this paper enables efficiency savings.

7 Conclusions

This paper has shown how the problem of checking that a goal is determinate can be reformulated as the problem of inferring a class of determinate goals. Despite the generality of this problem, this paper has shown how a determinate inference engine can be constructed by composing classic goal-independent success set analyses such as argument-size and depth-k analysis with modern backward analysis techniques. The paper has demonstrated that the analysis is tractable and the important of determinacy suggests that the analysis will be useful.

Acknowledgements This work was supported, in part, by NSF grants CCR-0131862 and INT-0327760. The authors are grateful to John Gallagher for gently guiding them through the binding-time analysis literature. The authors have also benefited from discussions on backward analysis with Samir Genaim.

References

[1] T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Two Classes of Boolean Functions for Dependency Analysis. *Science of Computer Programming*, 31(1):3–45, 1998.

[2] F. Benoy and A. King. Inferring Argument Size Relationships with CLP($\mathscr{R}$). In *Logic Program Synthesis and Transformation*, volume 1207 of *Lecture Notes in Computer Science*, pages 204–223. Springer-Verlag, 1997.

[3] 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.

[4] M. Bruynooghe, M. Leuschel, and K. Sagonas. A Polyvariant Binding-Time Analysis for Off-line Partial Deduction. In *European Symposium on Programming*, volume 1381 of *Lecture Notes in Computer Science*, pages 27–41, 2000.

[5] 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.

[6] S. K. Debray and N.-W. Lin. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, 1993.

[7] E. Eder. Properties of Substitutions and Unifications. *Journal of Symbolic Computation*, 1:31–46, 1985.

[8] J. P. Gallagher and L. Lafave. Regular Approximation of Computation Paths in Logic and Functional Languages. In O. Danvy, R. Glück, and P. Thiemann, editors, *International Seminar on Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 115–136. Springer-Verlag, 1996.

[9] S. Genaim and M. Codish. Inferring Termination Conditions for Logic Programs using Backwards Analysis. *Theory and Practice of Logic Programming*, To appear.

[10] R. Giacobazzi, S. Debray, and G. Levi. Generalized Semantics and Abstract Interpretation for Constraint Logic Programs. *The Journal of Logic Programming*, 25(3):191–248, 1995.

[11] R. Giacobazzi and L. Ricci. Detecting Determinate Computation by Bottom-Up Abstract Interpretation. In *European Symposium on Programming*, volume 582 of *Lecture Notes in Computer Science*, pages 167–181. Springer-Verlag, 1992.

[12] A. King and L. Lu. A Backward Analysis for Constraint Logic Programs. *Theory and Practice of Logic Programming*, 2:517–547, 2002.

[13] 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.

[14] L. Lu and A. King. Determinacy Inference for Logic Programs. Technical Report 19-04, Computing Laboratory, University of Kent, CT2 7NF, 2004. http://www.cs.kent.ac.uk/people/staff/amk/pubs.html.

[15] F. Mesnard and R. Bagnara. cTI: A constraint-based termination inference tool for ISO-Prolog. *Theory and Practice of Logic Programming*, To appear.

[16] R. A. O'Keefe. *The Craft of Prolog.* MIT Press, 1990.

[17] T. Sato and H. Tamaki. Enumeration of Success Patterns in Logic Programs. *Theoretical Computer Science*, 34:227–240, 1984.

[18] M. H. van Emden and R. A. Kowalski. The Semantics of Predicate Logic as a Programming Language. *Journal of the ACM*, 23(4):733–742, 1976.

8 Proof appendix

Let t be a term. The set of variables in t is denoted $vars(t)$. A substitution σ is a mapping from variables to terms such that $\mathrm{dom}(\sigma) = \{x \mid \sigma(x) \neq x\}$ is finite. The set $\mathrm{dom}(\sigma)$ is called the domain of σ. The range of σ is defined $rng(\sigma) = \bigcup_{x \in \mathrm{dom}(\sigma)} vars(t)$. The function composition operation $\circ$ is defined $g \circ f = \lambda x.g(f(x))$. A substitution σ is idempotent iff $\sigma \circ \sigma = \sigma$. The set of idempotent substitutions is denoted Sub. Let $\theta, \sigma \in \mathrm{Sub}$. We say that θ is less general than σ, denoted $\theta \leq \sigma$, iff there is an $\eta \in \mathrm{Sub}$ such that $\theta = \eta \circ \sigma$. Two substitutions θ and σ are said to be equivalent iff $\theta \leq \sigma$ and $\sigma \leq \theta$. The set of classes of equivalent substitutions with an added bottom *false* forms a complete lattice [7]. For the sake of simplicity, a class of equivalent substitutions is represented by one of its members. The partial order over the complete lattice is also written as $\leq$ and the greatest lower bound operator is denoted $\wedge$. Note that $\mathit{false} \leq \theta$ for any $\theta \in \mathrm{Sub}$. Two substitutions θ and σ are said to be consistent with each other iff $\theta \wedge \sigma \neq \mathit{false}$.

Let E be a set of equations. The set of most general unifiers of E is denoted $mgu(E)$. If E is not unifiable then $mgu(E) = \emptyset$. Member of $mgu(E)$ are equivalent to each other for a given set of equations E. We sometimes write $mgu(s,t)$ for $mgu(\{s = t\})$.

The operational semantics of logic programs is SLD derivation. In an SLD derivation, one goal is rewritten into another using a renaming instance of a clause until the empty goal is obtained or rewriting cannot be performed. Let σ be the most general unifier used in an earlier derivation step and θ the most general unifier used in a later derivation step. Then both the domain and the range of θ are disjoint with the domain of σ. The following lemma states that the composition of θ and σ is equivalent to their greatest lower bound.

Lemma 8.1 *Let* $\theta, \sigma \in \mathrm{Sub}$. *If* $(\mathrm{dom}(\theta) \cup rng(\theta)) \cap \mathrm{dom}(\sigma) = \emptyset$ *then* $\theta \circ \sigma = \theta \wedge \sigma$.

Proof 8.1 *Assume that* $(\mathrm{dom}(\theta) \cup rng(\theta)) \cap \mathrm{dom}(\sigma) = \emptyset$. *Let* $\eta = \{x \mapsto \theta(\sigma(x)) \mid x \in \mathrm{dom}(\sigma)\}$. *Then* $\mathrm{dom}(\eta) = \mathrm{dom}(\sigma)$ *and* $\theta \circ \sigma = \theta \cup \eta$. *We now prove that* $\theta \circ \sigma \leq \theta \wedge \sigma$. *By definition,* $\theta \circ \sigma \leq \sigma$. *Since* $\mathrm{dom}(\eta) = \mathrm{dom}(\sigma)$, *the assumption implies that* $\mathrm{dom}(\eta) \cap rng(\theta) = \emptyset$ *and*

$$\eta \circ \theta = \eta \cup \{y \mapsto \eta(\theta(y)) \mid y \in \mathrm{dom}(\theta)\} = \eta \cup \{y \mapsto \theta(y) \mid \mathrm{dom}(\theta) = \theta \circ \sigma\} = \eta \cup \theta$$

Thus, $\theta \circ \sigma \leq \theta$ *which together with* $\theta \circ \sigma \leq \sigma$ *implies that* $\theta \circ \sigma \leq \theta \wedge \sigma$.

We now complete the proof by showing that $\theta \wedge \sigma \leq \theta \circ \sigma$. By definition, $(\theta \wedge \sigma)(y) \leq \theta(y)$ for each $y \in \mathrm{dom}(\theta)$ and that $(\theta \wedge \sigma)(x) \leq \sigma(x)$ for each $x \in \mathrm{dom}(\sigma)$. Since $\theta \wedge \sigma \leq \theta$, $\mathrm{dom}(\theta \wedge \sigma) \supseteq \mathrm{dom}(\theta)$. Then $rng(\theta \wedge \sigma) \cap \mathrm{dom}(\theta) = \emptyset$ because $\theta \wedge \sigma$ is idempotent. Thus, $\theta((\theta \wedge \sigma)(x)) = (\theta \wedge \sigma)(x)$ for each $x \in \mathrm{dom}(\sigma)$ and $(\theta \wedge \sigma)(x) = \theta((\theta \wedge \sigma)(x)) \leq \theta(\sigma(x)) = \eta(x)$ for each $x \in \mathrm{dom}(\sigma)$. So, $(\theta \wedge \sigma) \leq \theta \cup \eta = \theta \circ \sigma$. ∎

Proof 8.2 (for proposition 3.1) *Let $\vec{x} = \vec{y}\vec{z}$. Let $p(\vec{x})$:-$c_i(\vec{x})$ be the success pattern for clause C_i.*

$$\gamma_{rigid'_\alpha}(\vec{y}) = \{\sigma \mid \forall y \in \vec{y}.rigid'_\alpha(\sigma(y))\}$$

and

$$\gamma_\alpha(c_i(\vec{y}\vec{z})) = \{\sigma \mid \quad \models c_i(\alpha(\sigma(\vec{y}))\alpha(\sigma(\vec{z})))\}$$

Then $\gamma_\alpha(c_i(\vec{y}\vec{z}))$ contains all the successful program states that can be reached from a derivation for $p(\vec{x})$ that commences with clause C_i; and $\gamma_{rigid'_\alpha}(\vec{y})$ contains all initial program states that make each $y \in \vec{y}$ rigid with respect to α.

The closure of a set of substitutions S with respect to (inverse) instantiation is defined $\uparrow S = \{\theta \mid \exists \sigma \in S.\theta \leq \sigma\}$ where $\leq$ is the more general than relation. The closure $\uparrow \gamma_\alpha(c_i(\vec{y}\vec{z}))$ thus contains all the initial program states for which there exists a successful derivation that commences with C_i.

We first prove that $(\uparrow \gamma_\alpha(c_i(\vec{y}\vec{z})) \cap \gamma_{rigid'_\alpha}(\vec{y})) \subseteq \gamma_\alpha(\exists_{\vec{z}}.c_i(\vec{y}\vec{z}))$. Let θ be an arbitrary substitution such that $\theta \in (\uparrow \gamma_\alpha(c_i(\vec{y}\vec{z}))) \cap \gamma_{rigid'_\alpha}(\vec{y}))$. Then $\theta(y)$ is rigid with respect to α for each $y \in \vec{y}$ and $\theta \in \uparrow \gamma_\alpha(c_i(\vec{y}\vec{z}))$. By the definition of $\uparrow$, there exists a substitution η such that $\eta \circ \theta \in \gamma_\alpha(c_i(\vec{y}\vec{z}))$. Hence, $\models c_i(\alpha(\eta \circ \theta(\vec{y}))\alpha(\eta \circ \theta(\vec{z})))$ by the definition of γ_α. Since $\theta(y)$ is rigid with respect to α for each $y \in \vec{y}$, $\alpha(\eta \circ \theta(\vec{y})) = \alpha(\theta(\vec{y}))$. Thus, $\models c_i(\alpha(\theta(\vec{y}))\alpha(\eta \circ \theta(\vec{z})))$ which implies that $\models \exists_{\vec{z}}.c_i(\alpha(\theta(\vec{y}))\vec{z})$. By the definition of γ_α, $\theta \in \gamma_\alpha(\exists_{\vec{z}}.c_i(\vec{y}\vec{z}))$. So, $(\uparrow \gamma_\alpha(c_i(\vec{y}\vec{z})) \cap \gamma_{rigid'_\alpha}(\vec{y})) \subseteq \gamma_\alpha(\exists_{\vec{z}}.c_i(\vec{y}\vec{z}))$.

We now prove the proposition by contradiction. Assume that some derivations of $\theta(p(\vec{x}))$ using C_1 as the first clause succeed and that some derivations of $\theta(p(\vec{x}))$ using C_2 as the first clause succeed. Then

$$\begin{array}{rcll}
\theta & \in & (\uparrow \gamma_\alpha(c_1(\vec{y}\vec{z}))) \cap (\uparrow \gamma_\alpha(c_2(\vec{y}\vec{z}))) \cap \gamma_{rigid'_\alpha}(\vec{y}) & \\
& = & (\uparrow \gamma_\alpha(c_1(\vec{y}\vec{z}))) \cap \gamma_{rigid'_\alpha}(\vec{y}) \ \cap \ (\uparrow \gamma_\alpha(c_2(\vec{y}\vec{z}))) \cap \gamma_{rigid'_\alpha}(\vec{y}) & \\
& \subseteq & \gamma_\alpha(\exists_{\vec{z}}.c_1(\vec{y}\vec{z})) \cap \gamma_\alpha(\exists_{\vec{z}}.c_2(\vec{y}\vec{z})) & \\
& = & \gamma_\alpha(\exists_{\vec{z}}.c_1(\vec{y}\vec{z}) \wedge \exists_{\vec{z}}.c_2(\vec{y}\vec{z})) & \text{by definition of } \gamma_\alpha \\
& = & \gamma_\alpha(\overline{\exists}_{\vec{y}}.c_1(\vec{x}) \wedge \overline{\exists}_{\vec{y}}.c_2(\vec{x})) & \text{by definition of } \overline{\exists}
\end{array}$$

$$
\begin{aligned}
&= \gamma_\alpha(\mathit{false}) && \text{since } \mathcal{X}_P(\vec{y}, p(\vec{x}), C_1, C_2) \text{ holds}\\
&= \emptyset && \text{by definition of } \gamma_\alpha
\end{aligned}
$$

where the set inequality holds because $(\uparrow \gamma_\alpha(c_i(\vec{y}\vec{z})) \cap \gamma_{rigid'_\alpha}(\vec{y})) \subseteq \gamma_\alpha(\exists_{\vec{z}}.c_i(\vec{y}\vec{z}))$. *Since* $\theta \in \emptyset$ *is false, the assumption is contradicted that there exist successful derivations of* $\theta(p(\vec{x}))$ *using* C_1 *as the first clause and successful derivations of* $\theta(p(\vec{x}))$ *using* C_2 *as the first clause. Therefore, the proposition holds.* ∎

Theorem 4.1. Let θ be substitution and θ satisfy $\mathcal{D}_P(p(\vec{x}))$. Then $\theta(p(\vec{x}))$ has at most one computed answer.

Proof 8.3 *Let* $I_n = D_P^n(\top)$ *where* $\top = \{p(\vec{x}) \mapsto \mathit{true} \mid p(\vec{x}) \text{ is defined in } P\}$. *It suffices to prove that* $\forall n \geq 0.Q(n)$ *holds where* $Q(n)$ *is defined*

> *For any* $\theta \in \mathit{Sub}$, *if* $\theta \in \gamma_{rigid'_\alpha}(I_n(p(\vec{x})))$ *then* $\theta(p(\vec{x}))$ *has at most one successful SLD derivation of length equal to or less than* n.

The proof is done by mathematical induction on n.

Basis. $Q(0)$ *holds vacuously since there is no successful SLD derivation of length* 0.

Induction. Assume that $Q(k)$ *holds for all* $k \leq n$. *It remains to prove that* $Q(n+1)$ *holds. Suppose that* $\theta \in \gamma_{rigid'_\alpha}(I_{n+1}(p(\vec{x})))$. *By the definition of* D_P, $\theta \in \gamma_{rigid'_\alpha}(\mathcal{X}_P(p(\vec{x})))$. *Thus, according to proposition 3.1, all successful derivations for* $\theta(p(\vec{x}))$, *if any, commences with a single clause* C. *Let* C *be* $p(\vec{x}) \leftarrow E, p_1(x_1), \cdots, p_m(x_m)$ *and* $\zeta \in mgu(E)$. *Recall that* $\gamma_{rigid'_\alpha}(\overline{\forall}_x(e)) \subseteq \gamma_{rigid'_\alpha}(e)$ *and that* $\zeta \in \gamma_{rigid'_\alpha}(f)$. *Since* $\theta \in \gamma_{rigid'_\alpha}(I_{n+1}(p(\vec{x})))$, $\theta \in \gamma_{rigid'_\alpha}(e_i)$ *where* $e_i = (f_i \Rightarrow I_n(p_i(\vec{x}_i)))$ *and* $f_i = f \wedge \bigwedge_{1 \leq j < i} \mathcal{R}_P(p(\vec{x}_j))$. *Let* σ *be a computed answer of* $\theta(p(\vec{x}))$ *resulting from a successful derivation of length equal to or less than* $n+1$. *Then* $\theta(p(\vec{x})) \overset{\sigma_0, C}{\Longrightarrow} \sigma_0 \circ \theta(p_1(\vec{x}_1), \cdots, p_m(\vec{x}_m))$ *with* $\sigma_0 \in mgu(\theta(E))$, $\sigma_j \circ \cdots \circ \sigma_0 \circ \theta(p_{j+1}(\vec{x}_{j+1}))$ *has a computed answer* σ_j *resulting from a successful derivation of length equal to or less than* n *for each* $0 \leq j < m$, *and* $\sigma = \sigma_m \circ \cdots \circ \sigma_0$. *Let* $\theta_i = \sigma_{i-1} \circ \cdots \circ \sigma_0 \circ \theta$. *Observe that* $\sigma_0 \in \gamma_{rigid'_\alpha}(f)$ *since* $\sigma_0 \leq \zeta$ *and* $\zeta \in \gamma_{rigid'_\alpha}(f)$. *Also we have that* $\sigma_i \in \gamma_{rigid'_\alpha}(\mathcal{R}_P(p_i(\vec{x}_i)))$. *By lemma 8.1,* $\theta_i = \sigma_{i-1} \wedge \cdots \wedge \sigma_0 \wedge \theta$. *Thus* $\theta_i \in \gamma_{rigid'_\alpha}(f \wedge \bigwedge_{1 \leq j < i} \mathcal{R}_P(p(\vec{x}_j)))$ *and hence* $\theta_i \in \gamma_{rigid'_\alpha}(I_n(p_i(\vec{x}_i)))$. *So, by the induction hypothesis, each* σ_i *is unique and hence* σ *is unique. Therefore,* $Q(n+1)$ *holds.* ∎