Goal-Independent Suspension Analysis for Logic Programs with Dynamic Scheduling

Samir Genaim* and Andy King

[1] Ben-Gurion University of the Negev, PoB 653, Beer-Sheva, 84105 Israel.
[2] University of Kent, Canterbury, CT2 7NF, UK.

Abstract. A goal-independent suspension analysis is presented that infers a class of goals for which a logic program with delays can be executed without suspension. The crucial point is that the analysis does not verify that an (abstract) goal does not lead to suspension but rather it infers (abstract) goals which do not lead to suspension.

1 Introduction

A logic program can be considered as consisting of a logic component and a control component [15]. Although the meaning of the program is largely defined by its logical specification, choosing the right control is crucial in obtaining a correct and efficient program. In recent years, one of the most popular ways of defining control is by suspension mechanisms which delay the selection of a sub-goal until some condition is satisfied [2]. Delays have proved to be invaluable for handling negation, delaying non-linear constraints, enforcing termination, improving search and modelling concurrency. However, reasoning about logic programs with delays is notoriously difficult and one reoccurring problem for the programmer is that of determining whether a given program and goal can reduce to a state which possesses a sub-goal that suspends indefinitely. A number of abstract interpretation schemes [3,5,8] have therefore been proposed for verifying that a program and goal cannot suspend in this fashion. These analyses essentially simulate the operational semantics tracing the execution of the program with collections of abstract states, and are thus said to be goal-*dependent*. This paper presents a suspension analysis that is performed in a goal-*independent* way. Specifically, rather than verifying that a *particular* goal will not lead to a suspension, the analysis infers a *class* of goals that will not lead to suspension. This new approach has the computational advantage that the programmer need not rerun the analysis for different (abstract) queries.

The analysis also tackles suspension analysis from another new perspective – it verifies whether a logic program with delays can be scheduled with a *local* selection rule [20]. Under local selection, the selected atom is completely resolved, that is, those atoms it directly and indirectly introduces are also resolved, before any other atom is selected. Leftmost selection is one example of

* New address: Universita' degli Studi di Verona, 37134 Verona, Italy.

local selection. Knowledge about suspension within the context of local selection is useful within it own right [8,14] but it turns out that local selection also fits elegantly with backward reasoning. Moreover, any program that can be shown to be suspension-free under local selection is clearly suspension-free with a more general selection rule (though the converse does not follow). Our analysis draws together a number of strands in program analysis and therefore, for clarity, we summarise our contribution:

– The analysis performs goal-independent suspension analysis.
– The analysis, though technical, reduces to two simple bottom-up fixpoint computations – a lfp and a gfp – which, like the backward analysis of [13], makes it simple to implement. The rôle of the lfp is simply to calculate success patterns that are used within the gfp calculation to model the way the sub-goals of a compound goal can bind variables.
– The analysis is straightforward like the simple but successful suspension framework of Debray *et al* [8] that infers suspension-freeness under leftmost selection. The analysis in this paper additionally considers *all* local selection rules and therefore strikes a good balance between tractability and precision.
– The analysis is unique in that it exploits the property that Heyting closed domains [11] possess a pseudo-complement for two effects. First, the pseudo-complement which enables information flow to be reversed to obtain a goal-independent analysis (this idea is not new [13]). Second, pseudo-complement is used to model synchronisation. The crucial correctness result exploits a (reordering) relationship between monotonic and positive Boolean functions and Boolean implication.

The paper is structured as follows: Section 2 presents an example that illustrates the ideas behind the analysis. Section 3 introduces the necessary preliminaries. Section 4 details local selection. Section 5 explains the rôle of Boolean functions in analysis. Section 6 details the analysis itself and Section 7 presents an experimental evaluation. Section 8 reviews related work and Section 9 concludes.

2 Worked Example

Consider the Prolog program listed in the left-hand column of Figure 1. Declaratively, the program defines the relation that the second argument (a list) is an in-order traversal of the first argument (a tree). Operationally, the declaration `:- block app(-,?,-)` delays (blocks) app goals until their arguments are sufficiently instantiated. The dashes in the first and third argument positions specify that a call to app is to be delayed until either its first or third argument are bound to non-variable terms. Thus app goals can be executed in one of two modes. The problem is to compute input modes which are sufficient to guarantee that any inorder query which satisfies the modes will not lead to a suspension under local selection. This problem can be solved with backward analysis. Backward analysis infers requirements on the input which ensure that certain properties hold at (later) program points [13]. The analysis reduces to three steps: a program abstraction step; least fixpoint (lfp) and a greatest fixpoint (gfp) computation.

```
inorder(nil,[]).
inorder(tree(L,V,R),I) :-
    app(LI,[V|RI],I),
    inorder(L,LI),
    inorder(R,RI).

:- block app(-, ?, -).
app([], X, X).
app([X|Xs], Ys, [X|Zs]) :-
    app(Xs,Ys,Zs).
```

```
inorder(T, I) :-
    true :
    T = nil, I = [] : true.
inorder(T, I) :-
    true :
    T = tree(L,V,R),
    A = [V|RI] :
    app(LI,A,I),
    inorder(L,LI),
    inorder(R,RI).

app(L, Ys, A) :-
    nonvar(L) ∨ nonvar(A):
    L = [], A = Ys : true.
app(L, Ys, A) :-
    nonvar(L) ∨ nonvar(A):
    L = [X|Xs], A = [X|Zs] :
    app(Xs,Ys,Zs).
```

```
inorder(T, I) :-
    true : T ∧ I : true
inorder(T, I) :-
    true :
    T ↔ (L ∧ V ∧ R),
    A ↔ (V ∧ RI) :
    app(LI,A,I),
    inorder(L,LI),
    inorder(R,RI).

app(L, Ys, A) :-
    L ∨ A :
    L ∧ (A ↔ Ys) : true.
app(L, Ys, A) :-
    L ∨ A :
    L ↔ (X ∧ Xs),
    A ↔ (X ∧ Zs) :
    app(Xs,Ys,Zs).
```

Fig. 1. inorder program in Prolog, in ccp and as a *Pos* abstraction

2.1 Program Abstraction

Abstraction in turn reduces to two transformations: one from a Prolog with delay program to a concurrent constraint programming (ccp) program and another from the ccp program to a *Pos* abstraction. The Prolog program is re-written to a ccp program to make blocking requirements explicit in the program as ask constraints. More exactly, a clause of a ccp program takes the form h :– c' : c'' : g where h is an atom, g is a conjunction of body atoms and c' and c'' are the ask and tell constraints. The asks are guards that inspect the store and specify synchronisation behaviour whereas the tells are single-assignment writes that update the store. Empty conjunctions of atoms are denoted by true. The nonvar(x) constraint states the requirement that x is bound to a non-variable term. The second transform abstracts the ask and tell constraints with Boolean functions which capture instantiation dependencies. The ask constraints are abstracted from below whereas the tell constraints are abstracted from above. More exactly, an ask abstraction is stronger than the ask constraint – whenever the abstraction holds then the ask constraint is satisfied; whereas an tell abstraction is weaker than the tell constraint – whenever the tell constraint holds then so does its abstraction. For example, the function L $\vee$ A describes states where either L or A is ground [1] which, in turn, ensure that the ask constraint nonvar(L) $\vee$ nonvar(A) holds. On the other hand, once the tell `A = [V|RI]` holds, then the grounding behaviour of the state (and all subsequent states) is described by A $\leftrightarrow$ (V $\wedge$ RI).

2.2 Least Fixpoint Calculation

The second step of the analysis approximates the success patterns of the ccp program (and thus the Prolog with delays program) by computing a lfp of the abstract *Pos* program. A success pattern is an atom with distinct variables for arguments paired with a *Pos* formula over those variables. A success pattern summarises the behaviour of an atom by describing the bindings it can make. The lfp of the *Pos* program can be computed T_P-style [10] in a finite number of iterates. Each iterate is a set of success patterns: at most one pair for each predicate in the program. This gives the following lfp:

$$F = \left\{ \begin{array}{l} \langle \text{inorder}(x_1, x_2), x_1 \leftrightarrow x_2 \rangle \\ \langle \text{app}(x_1, x_2, x_3), (x_1 \wedge x_2) \leftrightarrow x_3 \rangle \end{array} \right\}$$

Observe that F faithfully describes the grounding behaviour of inorder and app.

2.3 Greatest Fixpoint Calculation

A gfp is computed to characterise the safe call patterns of the program. A call pattern has the same form as a success pattern. Iteration commences with

$$D_0 = \left\{ \begin{array}{l} \langle \text{inorder}(x_1, x_2), \mathit{true} \rangle \\ \langle \text{app}(x_1, x_2, x_3), \mathit{true} \rangle \end{array} \right\}$$

and incrementally strengthens the call pattern formulae until they are safe, that is, they describe queries which are guaranteed not to violate the ask constraints. The iterate D_{i+1} is computed by putting $D_{i+1} = D_i$ and then revising D_{i+1} by considering each $p(\boldsymbol{x})$:- $d : f : p_1(\boldsymbol{x}_1), \ldots, p_n(\boldsymbol{x}_n)$ in the abstract program and calculating a (monotonic) formula that describes input modes (if any) under which the atoms in the clause can be scheduled without suspension under local selection. A monotonic formula over set of variables X is any formula of the form $\vee_{i=1}^{n}(\wedge Y_i)$ where $Y_i \subseteq X$ [7]. Let d_i denote a monotonic formula that describes the call pattern requirement for $p_i(\boldsymbol{x}_i)$ in D_i and let f_i denote the success pattern formula for $p_i(\boldsymbol{x}_i)$ in the lfp (that is not necessarily monotonic). A new call pattern for $p(\boldsymbol{x})$ is computed using the following algorithm:

- Calculate $e = \wedge_{i=1}^{n}(d_i \rightarrow f_i)$ that describes the grounding behaviour of the compound goal $p_1(\boldsymbol{x}_1), \ldots, p_n(\boldsymbol{x}_n)$. The intuition is that $p_i(\boldsymbol{x}_i)$ can be described by $d_i \rightarrow f_i$ since if the input requirements hold (d_i) then $p_i(\boldsymbol{x}_i)$ can be executed without suspension, hence the output must also hold (f_i).
- Compute $e' = \wedge_{i=1}^{n} d_i$ which describes a groundness property sufficient for scheduling all of the goals in the compound goal without suspension. Then $e \rightarrow e'$ describes a grounding property which, if satisfied, when the compound goal is called ensures the goal can be scheduled by local selection without suspension (this relies on an unusual reordering property of monotonic functions that is explained in Section 5.3).
- Calculate $g = d \wedge (f \rightarrow (e \rightarrow e'))$ that describes a grounding property which is strong enough to ensure that both the ask is satisfied and the body atoms can be scheduled by local selection without suspension.
- Eliminate those variables not present in $p(\boldsymbol{x})$, Y say, by computing $g' = \forall_Y(g)$ where $\forall_{\{y_1 \ldots y_n\}}(g) = \forall_{y_1}(\ldots \forall_{y_n}(g))$. A single variable can be

eliminated by $\forall_x(f) = f'$ if $f' \in \mathit{Pos}$ otherwise $\forall_x(f) = 0$ where $f' = f[x \mapsto 0] \wedge f[x \mapsto 1]$. Hence $\forall_x(f)$ entails f and g' entails g, so that a safe calling mode for this particular clause is then given by g'.

– Compute a monotonic function g'' that entails g'. Since g'' is stronger than g' it follows that g'' is sufficient for scheduling the compound goal by local selection without suspension. The function g' needs to be approximated by a monotonic function since the $e \to e'$ step relies on d_i being monotonic.
– Replace the pattern $\langle p(\boldsymbol{x}), g'''\rangle$ in D_{i+1} with $\langle p(\boldsymbol{x}), g'' \wedge g'''\rangle$.

This procedure generates the following D_i sequence:

$$D_1 = \left\{ \begin{array}{l} \langle \text{inorder}(x_1, x_2), \mathit{true}\rangle \\ \langle \text{app}(x_1, x_2, x_3), x_1 \vee x_3\rangle \end{array} \right\} \qquad D_2 = \left\{ \begin{array}{l} \langle \text{inorder}(x_1, x_2), x_1 \vee x_2\rangle \\ \langle \text{app}(x_1, x_2, x_3), x_1 \vee x_3\rangle \end{array} \right\}$$

The gfp is reached and checked in three iterations. The result asserts that a local selection rule exists for which inorder will not suspend if either its first or second arguments are ground. Indeed, observe that if the first argument is ground then body atoms of the second inorder clause can be scheduled as follows inorder(L,LI), then inorder(R,RI), and then app(LI,A,I) whereas if the second argument is ground, then the reverse ordering is sufficient for non-suspension.

3 Preliminaries

Let $\wp^+(S)$ (S^*) denote the set of multisets (sequences) whose elements are drawn from S. Let ϵ denote the empty sequence, let . denote sequence concatenation and let $\|s\|$ denote the length of a sequence s. If s is a sequence, let $\Pi(s)$ denote the set of permutations of s. Let $[l, u] = \{n \in \mathbb{Z} \mid l \leq n \leq u\}$. Transitive closure of a binary relation ϱ is denoted $\varrho^\star$.

3.1 Terms, Substitutions, and Equations

Let Term denote the set of (possibly infinite) terms over an alphabet of functor symbols Func and a (denumerable) universe of variables Var where $\mathit{Func} \cap \mathit{Var} = \emptyset$. Let $\mathit{var}(t)$ denote the set of variables occurring in the term t.

A substitution is a (total) map $\theta : \mathit{Var} \to \mathit{Term}$ such that $\mathit{dom}(\theta) = \{u \in \mathit{Var} \mid \theta(u) \neq u\}$ is finite. Let $\mathit{rng}(\theta) = \cup\{\mathit{var}(\theta(u)) \mid u \in \mathit{dom}(\theta)\}$ and let $\mathit{var}(\theta) = \mathit{dom}(\theta) \cup \mathit{rng}(\theta)$. A substitution θ is idempotent iff $\theta \circ \theta = \theta$, or equivalently, iff $\mathit{dom}(\theta) \cap \mathit{rng}(\theta) = \emptyset$. Let Sub denote the set of idempotent substitutions and let id denote the empty substitution. Let $\theta(t)$ denote the term obtained by simultaneously replacing each occurrence of a variable $x \in \mathit{dom}(\theta)$ in t with $\theta(x)$. An equation e is a pair $(s = t)$ where $s, t \in \mathit{Term}$. A finite set of equations is denoted E and Eqn denotes the set of finite sets of equations. Also define $\theta(E) = \{\theta(s) = \theta(t) \mid (s = t) \in E\}$. The map eqn : $\mathit{Sub} \to \mathit{Eqn}$ is defined $\text{eqn}(\theta) = \{x = \theta(x) \mid x \in \mathit{dom}(\theta)\}$. Composition $\theta \circ \psi$ of two substitutions is defined so that $(\theta \circ \psi)(u) = \theta(\psi(u))$ for all $u \in V$. Composition induces the (more general than) relation $\leq$ defined by $\theta \leq \psi$ iff there exists $\delta \in \mathit{Sub}$ such that $\psi = \delta \circ \theta$ which, in turn, defines the equivalence relation (variance) $\theta \approx \psi$ iff $\theta \leq \psi$ and $\psi \leq \theta$. Let Ren denote the set of invertible substitutions (renamings).

3.2 Most General Unifiers

The set of unifiers of E is defined by: $unify(E) = \{\theta \in Sub \mid \forall(s = t) \in E.\theta(s) = \theta(t)\}$. The set of most general unifiers (mgus) and the set of idempotent mgus (imgus) are defined: $mgu(E) = \{\theta \in unify(E) \mid \forall\psi \in unify(E).\theta \leq \psi\}$ and $imgu(E) = \{\theta \in mgu(E) \mid dom(\theta) \cap rng(\theta) = \emptyset\}$. Note that $imgu(E) \neq \emptyset$ iff $mgu(E) \neq \emptyset$ [16].

3.3 Logic Programs

Let Pred denote a (finite) set of predicate symbols, let *Atom* denote the set of (flat) atoms over Pred with distinct arguments drawn from Var, and let $Goal = \wp^*(Atom)$. A logic program P (with dynamic scheduling assertions) is a finite set of clauses w of the form $w = h \mathrel{:-} D : E : b$ where $h \in Atom$, $D \in \wp(Eqn)$ (the ask is a set of equations), $E \in Eqn$ (the tell is a single equation) and $b \in Goal$. An operational semantics (that ignores each D and therefore synchronisation) is defined in terms of the standard transition system:

Definition 1 (standard transition system). Given a logic program P, $\rightsquigarrow_P \subseteq (Goal \times Sub)^2$ is the least relation such that: $s = \langle g, \theta\rangle \rightsquigarrow_P \langle b.g', \delta \circ \theta\rangle$ if

- there exists $p(\boldsymbol{x}) \in g$
- and there exists $\rho \in Ren$ and $w \in \rho(P)$ such that $\mathrm{var}(w) \cap \mathrm{var}(s) = \emptyset$ and $w = p(\boldsymbol{y}) \mathrel{:-} D : E : b$
- and $\delta \in imgu(\{\theta(\boldsymbol{x}) = \boldsymbol{y}\} \cup E)$ and $g' = g \setminus \{p(\boldsymbol{x})\}$

Note that . denotes concatenation. The operational semantics is the transitive closure of the relation on (atomic) goals, that is, $\mathcal{O}(P) = \{\theta(p(\boldsymbol{x})) \mid \langle p(\boldsymbol{x}), id\rangle \rightsquigarrow_P^\star \langle \epsilon, \theta\rangle\}$. The following lemmas are useful in establishing the main result, theorem 1, and follow from the switching lemma [17, lemma 9.1].

Lemma 1. Let $\langle a.g, \theta\rangle \rightsquigarrow_P^i \langle \epsilon, \theta'\rangle$. Then $\langle a, \theta\rangle \rightsquigarrow_P^j \langle \epsilon, \psi\rangle$ and $\langle g, \psi\rangle \rightsquigarrow_P^k \langle \epsilon, \psi'\rangle$ where $i = j + k$ and $\theta' \approx \psi'$.

Lemma 2. Suppose $\langle g_1, \theta_1\rangle \rightsquigarrow_P^\star \langle g_2, \theta_2\rangle$ and $\theta_1 \approx \psi_1$. Then $\langle g_1, \psi_1\rangle \rightsquigarrow_P^\star \langle g_2, \psi_2\rangle$ where $\theta_2 \approx \psi_2$.

A fixpoint semantics of P (that again ignores synchronisation) can be defined in terms of an immediate consequences operator $\mathcal{F}_P$. Let $Base = \{\theta(a) \mid a \in Atom \wedge \theta \in Sub\}$ and $Int = \{I \subseteq Base \mid \forall a \in I.\forall\theta \in Sub.\theta(a) \in I\}$. Then $\langle Int, \subseteq, \cup, \cap, Base, \emptyset\rangle$ is a complete lattice.

Definition 2. Given a logic program P, the operator $\mathcal{F}_P : Int \rightarrow Int$ is defined:

$$\mathcal{F}_P(I) = \left\{\theta(h) \,\middle|\, \begin{array}{l} h \mathrel{:-} D : E : a_1, \ldots, a_m \in P \wedge \\ \theta \in unify(E) \ \wedge \ \theta(a_i) \in I \end{array}\right\}$$

The operator $\mathcal{F}_P$ is continuous and hence the fixpoint semantics for a program P can be defined as $\mathcal{F}(P) = \mathrm{lfp}(\mathcal{F}_P)$. The relationship between the operational and fixpoint semantics is stated below.

Theorem 1 (Partial correctness). $\mathcal{O}(P) \subseteq \mathcal{F}(P)$.

Although the fixpoint semantics is only partially correct – it does not consider synchronisation – it still provides a useful foundation for analysis since any safe (superset) abstraction of $\mathcal{F}(P)$ is also a safe approximation of $\mathcal{O}(P)$.

4 Local Selection

This section formalises the analysis problem, and in particular local selection, by introducing an operational semantics for logic programs which combines delay with local selection. A transition system is defined in terms of an augmented notion of state, that is, $State = \{susp\} \cup Goal \times Sub \cup Goal \times Goal \times Sub$.

Definition 3 (transition system for local selection with delay). Given a logic program P, $\twoheadrightarrow_P \subseteq State^2$ is the least relation such that:

- $s = \langle p(\boldsymbol{x}).g, \theta\rangle \twoheadrightarrow_P \langle b, g', \delta \circ \theta\rangle$ if
  - there exists $\rho \in Ren$ and $w \in \rho(P)$ such that $\text{var}(w) \cap \text{var}(s) = \emptyset$ and $w = p(\boldsymbol{y}) :\!- D : E : b$
  - and there exists $E' \in D$ and $\mu \in unify(E')$ such that $\mu(p(\boldsymbol{y})) = \theta(p(\boldsymbol{x}))$
  - and $\delta \in imgu(\{\theta(\boldsymbol{x}) = \boldsymbol{y}\} \cup E)$ and $g' = g \setminus \{p(\boldsymbol{x})\}$;
- $s = \langle p(\boldsymbol{y}).g, \theta\rangle \twoheadrightarrow_P susp$ if
  - there exists $\rho \in Ren$ and $w \in \rho(P)$ such that $\text{var}(w) \cap \text{var}(s) = \emptyset$ and $w = p(\boldsymbol{y}) :\!- D : E : b$
  - and $\mu(p(\boldsymbol{y})) \neq \theta(p(\boldsymbol{x}))$ for all $E' \in D$ and for all $\mu \in unify(E')$;
- $\langle b, g, \theta\rangle \twoheadrightarrow_P \langle b'.g, \theta\rangle$ if $b' \in \Pi(b)$.

Recall that . is concatenation and $\Pi(b)$ is the set of goals obtained by permuting of the sequence of body atoms b. These permuted body atoms ensure that the transition system considers each local selection rule rather than a particular local selection rule. The analysis problem can now be stated precisely: it is to infer a sub-class of states of the form $s = \langle p(\boldsymbol{x}), \theta\rangle$ such that if $s \rightsquigarrow_P^\star \langle \epsilon, \psi\rangle$ then $s \twoheadrightarrow_P^\star \langle \epsilon, \chi\rangle$ where $\psi \approx \chi$. Put another way, if the standard transition system produces a computed answer then a local selection rule exists that will produce a variant of that answer. The problem is non-trivial because local selection can bar derivations from occurring that arise in the standard transition system. The following proposition is an immediate consequence of this.

Proposition 1. $\mathcal{O}(P) \supseteq \{\theta(p(\boldsymbol{x})) \mid \langle p(\boldsymbol{x}), id\rangle \twoheadrightarrow_P^\star \langle \epsilon, \theta\rangle\}$.

5 Boolean Functions

This section reviews Boolean functions and their rôle in analysis, before moving to introduce new properties of Boolean functions that are particularly pertinent to suspension analysis. A Boolean function is a function $f : Bool^n \rightarrow Bool$ where $n \geq 0$ and $Bool = \{0, 1\}$. A Boolean function can be represented by a propositional formula over $X \subseteq Var$ where $|X| = n$. The set of propositional formulae over X is denoted by $Bool_X$. Boolean functions and propositional formulae are used interchangeably without worrying about the distinction. The convention of identifying a truth assignment with the set of variables M that it maps to 1 is also followed. Specifically, a map $\psi_X(M) : \wp(X) \rightarrow Bool_X$ is introduced defined by: $\psi_X(M) = (\wedge M) \wedge \neg(\vee(X \setminus M))$. Henceforth suppose X is finite.

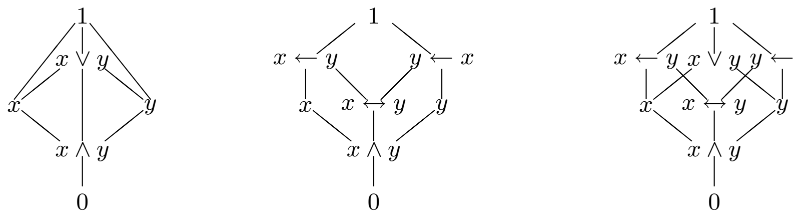

Fig. 2. Hasse diagrams for Mon_X, Def_X and Pos_X for the dyadic case $X = \{x, y\}$

Definition 4. The map $model_X : Bool_X \to \wp(\wp(X))$ is defined by: $model_X(f) = \{M \subseteq X \mid \psi_X(M) \models f\}$.

Example 1. If $X = \{x, y\}$, then the function $\{\langle 1,1\rangle \mapsto 1,\ \langle 1,0\rangle \mapsto 0,\ \langle 0,1\rangle \mapsto 0,\ \langle 0,0\rangle \mapsto 0\}$ can be represented by the formula $x \wedge y$. Moreover, $model_X(x \wedge y) = \{\{x, y\}\}$, $model_X(x \vee y) = \{\{x\}, \{y\}, \{x, y\}\}$, $model_X(false) = \emptyset$ and $model_X(true) = \wp(\wp(X)) = \{\emptyset, \{x\}, \{y\}, \{x, y\}\}$.

5.1 Classes of Boolean Functions

The suspension analysis is formulated with three classes of Boolean function.

Definition 5. A Boolean function f is positive iff $X \in model_X(f)$; f is definite iff $M \cap M' \in model_X(f)$ for all $M, M' \in model_X(f)$; f is monotonic iff $M' \in model_X(f)$ whenever $M \in model_X(f)$ and $M \subseteq M' \subseteq X$.

Let Pos_X denote the set of positive Boolean functions (augmented with 0); Def_X denote the set of positive functions over X that are definite (augmented with 0); and Mon_X denote the set of monotonic Boolean functions over X (that includes 0). Observe $Mon_X \subseteq Pos_X$ and $Def_X \subseteq Pos_X$. One useful representational property of Def_X is that if $f \in Def_X$ and $f \neq 0$, then $f = \wedge_{i=1}^{m}(y_i \leftarrow \wedge Y_i)$ for some $y_i \in X$ and $Y_i \subseteq X$ [7]. Moreover, if $f \in Mon_X$ and $f \neq 0$, then $f = \vee_{i=1}^{m}(\wedge Y_i)$ where $Y_i \subseteq X$ [6, Proposition 2.1]

The 4-tuple $\langle Pos_X, \models, \wedge, \vee\rangle$ is a finite lattice and Mon_X is a sub-lattice (whereas Def_X is not a sub-lattice as witnessed by the join of x and y in Figure 2). Existential quantification for Pos_X is defined by Schröder elimination, that is, $\exists x.f = f[x \mapsto 1] \vee f[x \mapsto 0]$. Universal projection is defined $\forall_x(f) = f'$ if $f' \in Pos_X$ otherwise $\forall_x(f) = 0$ where $f' = f[x \mapsto 0] \wedge f[x \mapsto 1]$. Note that $\exists x.(\exists y.f) = \exists y.(\exists x.f)$ and $\forall x.(\forall y.f) = \forall y.(\forall x.f)$ for all $x, y \in X$. Thus let $\exists\{y_1, \ldots, y_n\}.f = f_{n+1}$ where $f_1 = f$ and $f_{i+1} = \exists y_i.f_i$ and define $\forall\{y_1, \ldots, y_n\}.f$ analogously. Finally let $\overline{\exists} Y.f = \exists(X \backslash Y).f$ and $\overline{\forall} Y.f = \forall(X \backslash Y).f$.

5.2 Abstracting the Fixpoint Semantics Using Boolean Functions

Boolean functions are used to describe (grounding) properties of the program. The construction is to formalise the connection between functions and data (syntactic equations) and then extend it to semantic objects such as interpretations.

Definition 6. The abstraction $\alpha^{Pos} : \wp(Eqn) \to Pos$ and concretisation $\gamma^{Pos} : Pos \to \wp(Eqn)$ maps are defined:

$$\alpha^{Pos}(D) = \vee\{\alpha^{Def}(\theta) \mid \theta \in imgu(E) \wedge E \in D\} \quad \gamma^{Pos}(f) = \{E \mid \alpha^{Pos}(\{E\}) \models f\}$$

where $\alpha^{Def}(\theta) = \wedge\{x \leftrightarrow \mathrm{var}(t) \mid x \mapsto t \in \theta\}$.

The lifting of α^{Pos} and γ^{Pos} to interpretations is engineered so as to simplify the statement of the gfp operator, though it also suffices for defining the lfp operator. The construction starts with $Base^{Pos} = \{\langle a, f\rangle \mid a \in Atom \wedge f \in Pos_{\mathrm{var}(a)}\}$. To order these pairs, let $\boldsymbol{x} \leftrightarrow \boldsymbol{y} = \wedge_{i=1}^{n}(x_i \leftrightarrow y_i)$ where $\boldsymbol{x} = \langle x_1, \ldots, x_n\rangle$ and $\boldsymbol{y} = \langle y_1, \ldots, y_n\rangle$. The entailment order on Pos can be extended to $b_1, b_2 \in Base^{Pos}$ where $b_i = \langle p(\boldsymbol{x}_i), f_i\rangle$, $\mathrm{var}(\boldsymbol{x}) \cap \mathrm{var}(\boldsymbol{x}_i) = \emptyset$ and $f_i' = \exists var(\boldsymbol{x}_i).((\boldsymbol{x} \leftrightarrow \boldsymbol{x}_i) \wedge f_i)$ by defining $b_1 \models b_2$ iff $f_1' \models f_2'$. Observe that $\langle Base^{Pos}, \models\rangle$ is a pre-order since $\models$ is not reflexive. Equivalence on $Base^{Pos}$ is thus defined $b_1 \equiv b_2$ iff $b_1 \models b_2$ and $b_2 \models b_1$. Let $I_1, I_2 \subseteq Base^{Pos}/\equiv$. Then entailment lifts to $\wp(Base^{Pos}/\equiv)$ by $I_1 \models I_2$ iff for all $[b_1]_\equiv \in I_1$ there exists $[b_2]_\equiv \in I_2$ such that $b_1 \models b_2$.

Let Int^{Pos} denote the set of subsets I of $Base^{Pos}/\equiv$ such that there exists a *unique* $[\langle p(\boldsymbol{x}), f\rangle]_\equiv \in I$ for each $p \in Pred$. Since $Int^{Pos} \subseteq \wp(Base^{Pos}/\equiv)$, Int^{Pos} is also ordered by $\models$. Note, however, that $\models$ is the point-wise ordering on Int^{Pos} and that the lattice $\langle Int^{Pos}, \models, \vee, \wedge\rangle$ is equipped with simple $\vee$ and $\wedge$ operations. Specifically $\vee_{j\in J} I_j = \{[\langle p(\boldsymbol{x}), \vee_{j\in J} f_j\rangle]_\equiv \mid [\langle p(\boldsymbol{x}), f_j\rangle]_\equiv \in I_j\}$ and $\wedge_{j\in J} I_j$ is analogously defined. The following definition extends α^{Pos} and γ^{Pos} to interpretations and thereby completes the domain construction.

Definition 7. The concretisation map $\gamma^{Pos} : Int^{Pos} \to Int$ is defined:

$$\gamma^{Pos}(J) = \{\theta(a) \mid [\langle a, f\rangle]_\equiv \in J \wedge eqn(\theta) \in \gamma^{Pos}(f)\}$$

whereas $\alpha^{Pos} : Int \to Int^{Pos}$ is defined: $\alpha^{Pos}(I) = \wedge\{J \in Int^{Pos} \mid I \subseteq \gamma^{Pos}(J)\}$.

An operator that abstracts the standard fixpoint operator $\mathcal{F}_P$ is given below.

Definition 8. Given a logic program P, the fixpoint operator $\mathcal{F}_P^{Pos} : Int^{Pos} \to Int^{Pos}$ is defined by: $\mathcal{F}_P^{Pos}(I) = \wedge\{J \in Int^{Pos} \mid K \models J\}$ where

$$K = \left\{ [\langle h, f\rangle]_\equiv \;\middle|\; \begin{array}{cc} h \text{:-} D : E : a_1, \ldots, a_m \in P & \wedge \\ [\langle a_i, f_i\rangle]_\equiv \in I & \wedge \\ f = \overline{\exists}\mathrm{var}(h).(\alpha^{Pos}(\{E\}) \wedge \wedge_{i=1}^{m} f_i) & \end{array} \right\}$$

The operator $\mathcal{F}_P^{Pos}$ is continuous, hence an abstract fixpoint semantics is defined $\mathcal{F}^{Pos}(P) = \mathrm{lfp}(\mathcal{F}_P)$. The following correctness result is (almost) standard.

Theorem 2. $\mathcal{F}(P) \subseteq \gamma^{Pos}(\mathcal{F}^{Pos}(P))$.

5.3 Monotonic Boolean Functions

One idea behind the analysis is to use implication to encode synchronisation. The intuition is that if d_i expresses the required input and f_i the generated output for $p_i(\boldsymbol{x}_i)$, then $d_i \to f_i$ represents the behaviour of $p_i(\boldsymbol{x}_i)$. One subtlety is that $\wedge_{i=1}^{n}(d_i \to f_i)$ does not always correctly describe the behaviour of a compound goal $p_1(\boldsymbol{x}_1), \ldots, p_n(\boldsymbol{x}_n)$ if $d_i \notin Mon_X$. This is illustrated below.

Example 2. Consider the compound goal $p_1(x, y, z), p_2(x, y, z)$ for a two clause program $p_1(x, y, z) \mathrel{:-} D : z = c : \mathit{true}$ and $p_2(x, y, z) \mathrel{:-} \mathrm{nonvar}(x) : y = b : \mathit{true}$ where D is a (bizarre) ask constraint that is satisfied if y is ground whenever x is ground. Thus if $d_1 = (x \to y)$ and $d_2 = x$ hold then D and $\mathrm{nonvar}(x)$ are satisfied whereas $z = c$ and $y = b$ ensure that $f_1 = z$ and $f_2 = y$ hold. Neither $p_1(x, y, z)$ can be scheduled before $p_2(x, y, z)$ or vice versa to bind z, yet $\wedge_{i=1}^{2}(d_i \to f_i) \models z$. The problem stems from the implication in d_1. Ensuring that $d_i \in \mathit{Mon}_X$ avoids this problem as is formally asserted below.

Proposition 2. Let $f, f_i \in \mathit{Def}_X$ and $d_i \in \mathit{Mon}_X$ for all $i \in [1, m]$ and suppose $f \models (\wedge_{i=1}^{m}(d_i \to f_i)) \to (\wedge_{i=1}^{m} d_i)$. Then an injective map $\pi : [1, m] \to [1, m]$ exists such that $f \wedge \wedge_{j=1}^{j<i} f_{\pi(j)} \models d_{\pi(i)}$ for all $i \in [1, m]$.

The force of the result is that it states that the compound goal can be reordered as $p_{\pi(1)}(\boldsymbol{x}_{\pi(1)}), \ldots, p_{\pi(n)}(\boldsymbol{x}_{\pi(n)})$ so that the input requirement of goal $p_{\pi(i)}(\boldsymbol{x}_{\pi(i)})$ $(d_{\pi(i)})$ is satisfied by an initial binding (f) combined with those bindings output by the previous goals $(\wedge_{j=1}^{j<i} f_{\pi(j)})$. The following definitions explain how to (minimally) strengthen a positive function so as to obtain a monotonic function. The specification for this operation is captured in $\downarrow$.

Definition 9. The map $\downarrow : \mathit{Pos}_X \to \mathit{Mon}_X$ is defined $\downarrow f = \vee\{f' \in \mathit{Mon}_X \mid f' \models f\}$.

The operation $\downarrow$ arises during analysis and to construct a method for computing $\downarrow$, let $\rho : X \to X'$ be a bijective map where $X' \subseteq \mathit{Var}$ and $X \cap X' = \emptyset$. The proposition explains how $\circlearrowleft$ can be iteratively applied to finitely compute $\downarrow$.

Definition 10. The map $\circlearrowleft : \mathit{Pos}_X \to \mathit{Pos}_X$ is defined $\circlearrowleft f = \forall X'.f'$ where $f' = (\wedge_{i=1}^{n} x_i \to \rho(x_i)) \to \rho(f)$.

Proposition 3. Let $f \in \mathit{Pos}_X$. Then $\downarrow f = \wedge_{i \geq 1} f_i$ where $f_i \in \mathit{Pos}_X$ is the decreasing chain given by: $f_1 = f$ and $f_{i+1} = \circlearrowleft f_i$.

Example 3. Consider computing $\downarrow f$ where $X = \{x, y\}$ and $f = (x \to y)$. Suppose $\rho(x) = x'$ and $\rho(y) = y'$. Then $f' = ((x \to x') \wedge (y \to y')) \to (x' \to y')$, $f'[x' \mapsto 1] = (y \to y') \to y' = y \vee y'$ and $f'[x' \mapsto 0] = 1$ so that $\forall x'.f' = y \vee y'$. Put $f'' = y \vee y'$. Then $f''[y' \mapsto 1] = 1$ and $f''[y' \mapsto 0] = y$ so that $\circlearrowleft f = \forall y'.\forall x'.f' = \forall y'.f'' = y$. In fact $\circlearrowleft y = y$ so that $\downarrow f = y$. Observe that $y \models f$.

6 Suspension Analysis

This section draws together the previous sections to define the suspension analysis in terms of a backward fixpoint operator. To construct this operator, and specifically model asks, it is necessary to introduce a map $\alpha_{low}^{Pos} : \wp(\mathit{Eqn}) \to \mathit{Pos}$ that returns a lower approximation to a set of equations D. Recall that α^{Pos} yields an upper approximation in that if $E \in D$, then $\alpha^{Pos}(\{E\})$ entails $\alpha^{Pos}(D)$. Conversely α_{low}^{Pos}, which is defined below, delivers a lower approximation with the property that if $\alpha^{Pos}(\{E\})$ entails $\alpha_{low}^{Pos}(D)$, then $E \in D$.

Definition 11. The (lower) abstraction map $\alpha^{Pos}_{low} : \wp(Eqn) \rightarrow Pos$ is defined by: $\alpha^{Pos}_{low}(D) = \vee\{f \in Pos \mid \gamma^{Pos}(f) \subseteq D\}$.

Example 4. Let nonvar(x) and ground(y) denote the equation sets $\{E \in Eqn \mid \theta \in imgu(E) \wedge \theta(x) \notin Var\}$ and $\{E \in Eqn \mid \theta \in imgu(E) \wedge var(\theta(y)) = \emptyset\}$. Then $\alpha^{Pos}_{low}(Eqn) = 1$, $\alpha^{Pos}_{low}(\text{nonvar}(x)) = x$, $\alpha^{Pos}_{low}(\text{nonvar}(x) \cup \text{ground}(y)) = x \vee y$ and $\alpha^{Pos}_{low}(\{x = f(a)\}) = \alpha^{Pos}_{low}(\{x = y\}) = 0$.

Suspension analysis can now be formalised with an abstract fixpoint operator:

Definition 12. Given a logic program P, the operator $\mathcal{B}_P : Int^{Pos} \rightarrow Int^{Pos}$ is defined: $\mathcal{B}_P(I) = \vee\{J \in Int^{Pos} \mid \forall [b_1]_\equiv \in K. \exists [b_2]_\equiv \in J. b_2 \models b_1\}$ where

$$K = \left\{ [\langle h, d'' \rangle]_\equiv \;\middle|\; \begin{array}{lr} h :\!-D : E : a_1, \ldots, a_m \in P & \wedge \\ [\langle a_i, f_i \rangle]_\equiv \in \mathcal{F}^{Pos}(P) \;\wedge\; [\langle a_i, d_i \rangle]_\equiv \in I & \wedge \\ d \;=\; \alpha^{Pos}_{low}(D) \;\wedge\; e = \alpha^{Pos}(\{E\}) & \wedge \\ d' \;=\; (\wedge_{i=1}^m (d_i \rightarrow f_i)) \rightarrow (\wedge_{i=1}^m d_i) & \wedge \\ d'' \;=\; \downarrow(\overline{\forall}\text{var}(h).(d \wedge (e \rightarrow d'))) & \end{array} \right\}$$

Recall that $\wedge_{i=1}^m (d_i \rightarrow f_i)$ captures the grounding behaviour of the goal $a_1, \ldots, a_m$ whereas $\wedge_{i=1}^m d_i$ describes a state with variables sufficiently bound to enable each a_i to be scheduled with local selection without suspension. The function d' is a grounding property that, if satisfied when $a_1, \ldots, a_m$ is called, guarantees that $a_1, \ldots, a_m$ can be reordered so that each a_i can be scheduled by local selection without suspension. The function d'' is monotonic, defined only over those variables in h, and is sufficient to ensure that both the ask is satisfied and that $a_1, \ldots, a_m$ can be scheduled by local selection without suspension. If P contains a predicate p defined over n clauses, then $\{[p(\boldsymbol{x}, f_i)]_\equiv\}_{i=1}^n \subseteq K$ so in general $K \notin Int^{Pos}$. However, $\mathcal{B}_P(I)$ contains a *unique* element $[p(\boldsymbol{x}, f)]_\equiv$ such that $f = \wedge_{i=1}^n f_i$. In effect, related elements of K are merged with meet.

$\mathcal{B}_P$ is co-continuous and since Int^{Pos} is a finite lattice, it follows that gfp$(\mathcal{B}_P)$ exists. The value of gfp$(\mathcal{B}_P)$ is explained by the following theorem (or rather its corollary). It states that gfp$(\mathcal{B}_P)$ characterises a set of initial states for which if the standard transition system leads to a computed answer (in k steps) then local selection with delay leads to a variant of that computed answer (in k steps).

Theorem 3. *Suppose* $\theta(p(\boldsymbol{x})) \in \gamma^{Pos}(\mathcal{B}^k_P(\top))$, $s_1 = \langle p(\boldsymbol{x}), \theta \rangle$, $s_1 \rightsquigarrow^k_P \langle \epsilon, \psi \rangle$. *Then* $s_1 \twoheadrightarrow^k_P \langle \epsilon, \chi \rangle$ *where* $\psi \approx \chi$.

Corollary 1. *Suppose* $\theta(p(\boldsymbol{x})) \in \gamma^{Pos}(\text{gfp}(\mathcal{B}_P))$, $s_1 = \langle p(\boldsymbol{x}), \theta \rangle$, $s_1 \rightsquigarrow^k_P \langle \epsilon, \psi \rangle$. *Then* $s_1 \twoheadrightarrow^k_P \langle \epsilon, \chi \rangle$ *where* $\psi \approx \chi$.

To emphasise the significance of gfp$(\mathcal{B}_P)$, the abstract backward semantics for P is defined $\mathcal{B}(P) = \text{gfp}(\mathcal{B}_P)$. Co-continuity enables $\mathcal{B}(P)$ to be computed by *lower* Kleene iteration, that is, as the limit of $\top$, $\mathcal{B}_P(\top)$, $\mathcal{B}^2_P(\top)$, ... where $\top = \{[\langle p(\boldsymbol{x}), 1 \rangle]_\equiv \mid p \in Pred\}$. The example illustrates how to handle builtins.

Example 5. Consider the temperature conversion program in the left column of Fig. 3 which converts Celsius to Fahrenheit and vice versa. The `block` declaration equates to the equation set $D = (\text{nonvar}(X) \cap \text{nonvar}(Y)) \cup (\text{nonvar}(X) \cap$

```
cf(C, F) :- mul(C, 1.8, S), add(S, 32, F).

:- block add(-, -, ?), add(-, ?, -), add(?, -, -).
add(X, Y, Z) :- ground(X+Y), Z is X+Y.
add(X, Y, Z) :- ground(Z-X), Y is Z-X.
add(X, Y, Z) :- ground(Z-Y), X is Z-Y.

:- block mul(-, -, ?), mul(-, ?, -), mul(?, -, -).
mul(X, Y, Z) :- ground(X*Y), Z is X*Y.
mul(X, Y, Z) :- ground(Z/X), Y is Z/X.
mul(X, Y, Z) :- ground(Z/Y), X is Z/Y.
```

```
cf(C, F) :- true : T1 ∧ T2 :
   mul(C, T1, S), add(S, T2, F).

add(X, Y, Z) :- f : T ↔ (X ∧ Y) :
   ground(T), is(Z, T).
add(X, Y, Z) :- f : T ↔ (X ∧ Z) :
   ground(T), is(Y, T).
add(X, Y, Z) :- f : T ↔ (Y ∧ Z) :
   ground(T), is(X, T).

mul(X, Y, Z) :- ...

ground(X) :- true : X : true.
is(X, Y) :- true : X ∧ Y : true.
```

Fig. 3. conv program in Prolog and in *Pos* where $f = (\mathrm{X} \wedge \mathrm{Y}) \vee (\mathrm{X} \wedge \mathrm{Z}) \vee (\mathrm{Y} \wedge \mathrm{Z})$

nonvar$(Z)) \cup (\text{nonvar}(Y) \cap \text{nonvar}(Z))$ and $\alpha^{Pos}_{low}(D) = f$ (see Fig. 3). Note how the builtins ground and is are modelled in the abstract version of conv listed in the right column. For brevity, let $\boldsymbol{y} = \langle x_1, x_2 \rangle$ and $\boldsymbol{z} = \langle x_1, x_2, x_3 \rangle$. Then

$$\mathcal{F}^{Pos}(conv) = \left\{ \begin{array}{c} [\langle \text{cf}(\boldsymbol{y}),\ x_1 \wedge x_2 \rangle]_{\equiv} \\ [\langle \text{add}(\boldsymbol{z}),\ x_1 \wedge x_2 \wedge x_3 \rangle]_{\equiv} \\ [\langle \text{mul}(\boldsymbol{z}),\ x_1 \wedge x_2 \wedge x_3 \rangle]_{\equiv} \\ [\langle \text{ground}(x_1),\ x_1 \rangle]_{\equiv} \\ [\langle \text{is}(\boldsymbol{y}),\ x_1 \wedge x_2 \rangle]_{\equiv} \end{array} \right\} \quad K = \left\{ \begin{array}{r} [\langle \text{cf}(\boldsymbol{y}),\ 1 \rangle]_{\equiv} \\ [\langle \text{add}(\boldsymbol{z}),\ f \rangle]_{\equiv} \\ [\langle \text{mul}(\boldsymbol{z}),\ f \rangle]_{\equiv} \\ [\langle \text{ground}(x_1),\ 1 \rangle]_{\equiv} \\ [\langle \text{is}(\boldsymbol{y}),\ 1 \rangle]_{\equiv} \end{array} \right\}$$

where $f = (x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_2 \wedge x_3)$. Hence $\mathcal{B}_{conv}(\top) = K$. Then $\mathcal{B}^2_{conv}(\top)$ differs from $\mathcal{B}_{conv}(\top)$ only in $[\langle \text{cf}(\boldsymbol{y}), x_1 \vee x_2 \rangle]_{\equiv}$. In fact $\mathcal{B}(conv) = \mathcal{B}^2_{conv}(\top)$.

7 Experimental Evaluation

To assess the value of the analysis it has been implemented in SICStus Prolog using the BDD package of Armstrong and Schachte [1]. The implementation consists of two meta-interpreters – one for each fixpoint. Each abstract clause h :– d : f : $b_1, \ldots, b_n$ is represented as two facts: `my_clause`(h,[$id_f, b_1, \ldots, b_n$]) and `assertion`(h,id_d) where id_f and id_d are identifiers for the BDDs of f and d. Facts of the form `fact(gr`,$p(\boldsymbol{x})$,id_f`)` and `fact(ba`,$p(\boldsymbol{x})$,id_g`)` are added and removed from the database to record the status of the lfp and then the gfp. Both fixpoint engines are realised as semi-naive meta-interpreters.

The analyser has been applied to a number of programs: bestpath, entails, fact, hamming, inorder, isotrees, pascal, mm, hanoi, msort, qsort, queens, sieve, most of which derive from the Super Monaco benchmark suite. All programs were analysed in less than 1 second on a 500MHZ, 512MB Pentium III running RedHat Linux 7.2 with Kernel 2.4.7-10. The Super Monaco programs are coded in kl1 –

an early ccp language – and therefore for analysis these programs were manually translated into SICStus Prolog with blocks. It was for these programs that the analysis occasionally produced unexpected results (modes) and close inspection revealed errors in the hand translation. Some errors were straightforward (`block` declarations of the wrong arity) and other were subtle, but none came to light in the testing, presumably because of the particular interleaving adopted by the SICStus scheduler. These results suggest that the analysis has a rôle in bug detection. The analysis also inferred non-trivial modes for all predicates except for 6 mutually recursive predicates in bestpath for which false was returned. It is not yet clear whether a local selection rule exists for these predicates that avoids suspension – the synchronisation is subtle and may even be buggy. What is clear, however, is that local selection is sufficient to infer useful modes for the vast majority of the predicates that were analysed. An experimental analyser can be found at `http://www.cs.bgu.ac.il/cgi-bin/genaim/susweb.cgi` and the benchmarks are available from the home page of the second author.

8 Related Work

One of the most closely related works comes surprisingly from the compiling control literature and in particular the problem of *generating* a local selection rule under which a program universally terminates [12]. The technique of [12] builds on the termination inference method of [19] which infers initial modes for a query that, if satisfied, ensure that a logic program left-terminates. The chief advance in [12] over [19] is that it additionally infers how goals can be statically reordered so as to improve termination behaviour. This is performed by augmenting each clause with body atoms $a_1, \ldots, a_n$ with $n(n-1)/2$ Boolean variables $b_{i,j}$ with the interpretation that $b_{i,j} = 1$ if a_i precedes a_j in the reordered goal and $b_{i,j} = 0$ otherwise. The analysis of [19] is then adapted to include consistency constraints among the $b_{i,j}$, for instance, $b_{j,k} \wedge \neg b_{i,k} \Rightarrow \neg b_{i,j}$. In addition, the $b_{i,j}$ are used to determine whether the post-conditions of a_i contribute to the pre-conditions of a_j. Although motivated differently and realised differently (in terms of the Boolean μ-calculus) this work also uses Boolean functions to finesse the problem of enumerating the goal reorderings. This work complements our own since termination is a related but orthogonal requirement to non-suspension.

King and Lu [13] show how to apply backward analysis to the problem of figuring how to query a logic program with fixed selection rule. The analysis traces control-flow of the program (backward) right-to-left to infer the modes in which a predicate must be called under the leftmost selection rule. Although this analysis can be reinterpreted as a suspension analysis it cannot reason about local selection accurately since it only considers leftmost selection.

The early work of [5] presents an and-or tree framework that applies local reexecution to simulate the dataflow under different interleavings. A more direct approach is to abstract each state in the transition system with an abstract state to obtain an abstract transition system [3]. Finiteness is enforced through a widening known as star-abstraction [3]. This approach achieves a degree of conceptual simplicity though the abstract states themselves can be unwieldy. The work of [8] is unusual in that it attempts to detect suspension-freeness for goals under leftmost selection. Although this approach only considers one local selection rule, it is surprising effective because of the way data often flows left-to-right. A particularly elegant approach to suspension analysis follows from a confluence semantics that approximates the standard semantics in the sense that suspension implies suspension in the confluent semantics [4]. The crucial point is that because of confluence, an analysis based on the confluence semantics need only consider one scheduling rule. None of these analyses, however, can infer initial queries that guarantee non-suspension – all check for non-suspension. Other works have proposed generic abstract interpretation frameworks for dynamic scheduling [9,18] but none of these are for goal-independent analysis.

9 Concluding Discussion

This paper has shown how suspension analysis can be tackled for a new perspective – that of goal-independence. It shows how an analysis for non-suspension under local selection can be formulated as two simple bottom-up fixpoint computations. The analysis strikes a good balance between tractability and precision. It avoids the complexity of goal interleaving by exploiting reordering properties of monotonic and positive Boolean functions.

For reasons of presentation, the analysis proposed in this paper has been specified for logic programs. To further simplify the presentation, the analysis was formulated in terms of simple groundness dependencies. The first constraint can be relaxed by following a standard constraint formulation [10]. The second can be relaxed by lifting the analysis to rigidity (type) dependencies using term extractor maps [3,10]. Another direction for future work will be to generalise the analysis to other abstract domains that possess a pseudo-complement.

Acknowledgements. EU working group 23677 and EPSRC grant GR/MO8769 funded Samir Genaim while he visited the University of Kent. We thank Jacob Howe, Fred Mesnard and Ulf Nilsson for their valuable comments, and Bart Massey and Evan Tick for the Super Monaco benchmarks.

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. M. Carlsson. Freeze, Indexing, and Other Implementation Issues in the WAM. In *International Conference on Logic Programming*, pages 40–58. MIT Press, 1987.
3. M. Codish, M. Falaschi, and K. Marriott. Suspension Analyses for Concurrent Logic Programs. *Transactions on Programming Languages and Systems*, 16(3):649–686, 1994.

4. M. Codish, M. Falaschi, K. Marriott, and W. H. Winsborough. A Confluent Semantic Basis for the Analysis of Concurrent Constraint Logic Programs. *The Journal of Logic Programming*, 30(1):53–81, 1997.
5. C. Codognet, P. Codognet, and M. Corsini. Abstract Interpretation for Concurrent Logic Languages. In *North American Conference on Logic Programming*, pages 215–232. MIT Press, 1990.
6. A. Cortesi, G. Filé, and W. H. Winsborough. Optimal Groundness Analysis Using Propositional Logic. *The Journal of Logic Programming*, 27(2):137–167, 1996.
7. P. Dart. On Derived Dependencies and Connected Databases. *The Journal of Logic Programming*, 11(1–2):163–188, 1991.
8. S. Debray, D. Gudeman, and P. Bigot. Detection and Optimization of Suspension-free Logic Programs. *Journal of Logic Programming*, 29(1–3):171–194, 1992.
9. M. García de la Banda, K. Marriott, and P. J. Stuckey. Efficient Analysis of Logic Programs with Dynamic Scheduling. In *International Symposium on Logic Programming*, pages 417–431. MIT Press, 1995.
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 F. Scozzari. A Logical Model for Relational Abstract Domains. *Transactions on Programming Languages and Systems*, 20(5):1067–1109, 1998.
12. S. Hoarau and F. Mesnard. Inferring and Compiling Termination for Constraint Logic Programs. In *Logic-based Program Synthesis and Transformation*, volume 1559 of *Lecture Notes in Computer Science*, pages 240–254. Springer-Verlag, 1999.
13. A. King and L. Lu. A Backward Analysis for Constraint Logic Programs. *Theory and Practice of Logic Programming*, 2(4–5):517–547, 2002.
14. A. King and P. Soper. Schedule Analysis of Concurrent Logic Programs. In *Joint International Conference and Symposium on Logic Programming*, pages 478–492. MIT Press, 1992.
15. R. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, 1979.
16. J.-L. Lassez, M. Maher, and K. Marriott. Unification Revisited. In *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann, 1988.
17. J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 1993.
18. K. Marriott, M. García de la Banda, and M. Hermenegildo. Analyzing Logic Programs with Dynamic Scheduling. In *Principles of Programming Languages*, pages 240–254. ACM Press, 1994.
19. F. Mesnard. Inferring Left-terminating Classes of Queries for Constraint Logic Programs by means of Approximations. In *Joint International Conference and Symposium on Logic Programming*, pages 7–21. MIT Press, 1996.
20. L. Vielle. Recursive Query Processing: The Power of Logic. *Theoretical Computer Science*, 69(1):1–53, 1989.