Computer Science at Kent

Inferring Non-Suspension Conditions for Logic Programs with Dynamic Scheduling

Samir Genaim and Andy King

Technical Report No. 20-04
November 2004

Abstract A logic program consists of a logic component and a control component. The former is a specification in predicate logic whereas the latter defines the order of sub-goal selection. The order of sub-goal selection is often controlled with delay declarations that specify that a sub-goal is to suspend until some condition on its arguments is satisfied. Reasoning delay declarations is notoriously difficult for the programmer and it is not usual for a program and goal to reduce to a state that contains a sub-goal that suspends indefinitely. Suspending sub-goals are usually unintended and often indicate an error in the logic or control. A number of abstract interpretation schemes have therefore been proposed for checking that a given program and goal cannot reduce to such a state. This paper considers a reversal of this problem, advocating an analysis that for a given program infers a class of goals that do not lead to suspension. This paper shows that this more general approach can have computational, implementational and user-interface advantages. In terms of user-interface, this approach leads to a lightweight point-and-click mode of operation in which, after directing the analyser at a file, the user merely has to inspect the results inferred by the analysis. In terms of implementation, the analysis can be straightforwardly realised as two simple fixpoint computations. In terms of computation, by modelling $n!$ different scheduling of n sub-goals with a single Boolean function, it is possible to reason about the suspension behaviour of large programs. In particular, the analysis is fast enough to be applied repeatedly within the program development cycle. The paper also demonstrates that the method is precise enough to locate bugs in existing programs.

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

1 Introduction

A logic program can be considered as consisting of a logic component and a control component [34]. 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 [5, 44]. Delays have proved to be invaluable for handling negation [13], delaying non-linear constraints [31], enforcing termination [45], improving search [6] and supporting concurrency [51]. 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 contains a sub-goal that suspends indefinitely. States with suspending sub-goals are usually unintended and often indicate an error in the logic or control.

A number of abstract interpretation schemes [8, 9, 10, 16] have thus been proposed for checking that a program and goal cannot reduce to such a suspension state. These schemes simulate the operational semantics by tracing the execution of the program over a finite (though possibly large) collection of abstract states. Each abstract state describes a possibly infinite set of concrete states and because of the approximation inherent in abstract interpretation [12], these schemes either return "yes" – the program and goal definitely cannot reduce to a suspension state; or "don't know" – program and goal may reduce to a suspension state. In the former case, the program is verified as being (partially) correct, whereas in the latter case, it is potentially buggy and thus requires further inspection.

Ideally, automatic verification should be applied each time the program is altered in a non-trivial way. However, it is arguable that a programmer will routinely and frequently apply analysis only if it costs little. Specifically, the analysis should cost little in terms of human interaction; the programmer should be able to activate the analysis by merely pressing a button. Furthermore, if the programmer needs to scrutinise the results, then the results should be easy to interpret and relate directly to structural components of the program. A more subtle human interaction issue relates to when the analysis returns "don't know". In addition to engineering an analysis so that it is precise enough to be useful (does not yield too many false positives), another human interaction issue is to design an analysis around a conceptual model that can be grasped by the programmer. In the case of a "don't know", the programmer needs to appreciates why the analysis failed to return "yes" to decide whether the "don't know" results from a bug in the program or a lack of precision in the analysis. Finally, the analysis should also cost little in terms of time; programmer should not hesitate about applying the analysis even to the largest programs. Indeed, it can be argued that programs which contain the most concurrency, are likely to benefit the most from suspension analysis.

These are demanding requirements for any analysis, let alone one that aspires to reason about synchronisation. Not surprisingly, the previously proposed suspension analyses [8, 9, 10, 16] fail to satisfy these pragmatic requirements. Firstly, the and-or tree framework of [10] that applies a form of reexecution [36] to simulate the dataflow between the body atoms under different interleavings. And-or trees are abstraction devices [3] rather than programming or scheduling concepts and therefore it is not clear the extent to which a programmer will understand the underlying analysis model. On the other hand, the suspension analysis schemes that attain the required degree of conceptual simplicity [8, 9], do so by modelling the transition system induced by the program and goal with an abstract transition system between a finite number of abstract states. Each abstract state is a set of sub-goals paired with a set of dependencies between the variables occurring in the sub-goals. Since each abstract state can include one sub-goal for each predicate that is defined in the program, both the size and number of abstract states can grow large (even with widening [8]) and the practicality of these schemes has yet to be demonstrated. By way of contrast, the work of [16] is pragmatic in that it attempts to detect non-suspension by considering just one scheduling scheme: leftmost selection. Under leftmost selection, the leftmost sub-goal in a sequence of sub-goals is only considered for reduction. If the leftmost sub-goal can always be reduced and this, in turn, introduces sub-goals which can always be reduced when they are leftmost, then the whole goal can be verified as being suspension-free. This simple approach is robust enough to be engineered into an optimising compiler and is surprising effective because of the way data often flows left-to-right between the sub-goals of a compound goal [16].

This paper is motivated by the elegance of the suspension analyses based on abstract transition systems [8] and the pragmatism of the suspension analyses based on leftmost selection [16]. The new twist that this paper brings to suspension analysis is that rather than check that a particular goal will not lead to a suspension state, the analysis discovers (in a single application) a *class* of goals that will not lead to suspension. (This dichotomy between checking and inference is elsewhere described in terms of the goal-*dependent* versus goal-*independent* approach to analysis; checking is dependent on an initial goal whereas inference does not require an initial goal to be specified.) This advance owes much to research on domain refinement [17]. In particular, one way to enrich an abstract domain is to apply Heyting completion [24]. For example, by applying Heyting completion to the most basic groundness domain, *Con* [41], the domain of groundness dependencies, *Pos* [1], can be obtained. *Con* can merely express *conjunctions* such as $x \wedge y$ which is interpreted as stating that the program variables x and y are both bound to ground terms. On the other hand, *Pos* can express *dependencies* such as $x \rightarrow (y \wedge z)$ which encodes that whenever x is ground, then both y and z are ground. Heyting completion is more than a way of improving precision; the structure of a resulting domain enables information flow to be reversed during analysis [33]. Rather than verify that a property holds for a given query, it is possible to infer a class of queries under which the property holds.

Rather unusually, the analysis described in this paper also exploits a property of a domain that is closed under disjunctive completion [18]. The disjunctive completion of *Con* is *Mon*; the domain of monotonic functions which includes formulae such as $x \vee y$ which records that either x or y (or both) are ground. It also includes functions such as $x \wedge (y \vee z)$ and $w \wedge (x \vee (y \wedge z))$. This paper shows that the *Mon* domain, when used in concert with the *Pos* domain, has the important (and previously unexploited) computational attribute that it can be used to model synchronisation. The key idea is to model the $n!$ different scheduling of n sub-goals with a *single* Boolean function. To do so, each sub-goal is modelled by two functions: one in *Mon* and another drawn from *Pos*. The *Mon* function expresses groundness properties that are sufficient for non-suspension of the sub-goal. The *Pos* function captures the grounding behaviour of the sub-goal; it expresses grounding dependencies between the bindings made by the sub-goal. By appropriately composing these functions, it is possible to derive a function, say f, that describes grounding requirements that, if satisfied when the conjunction of sub-goals is called, is sufficient for a reordering of the sub-goals to exist that does not incur a suspension. Moreover, Boolean functions can be represented densely with binary decision diagrams (BDDs) and only $O(n)$ logical operations are needed to compute f. This approach leads to an efficient way of reasoning about the suspension behaviour of large programs.

As well as it computational properties, *Mon* is an intuitive domain for reporting non-suspension properties to a program developer. The domain in some sense is not alien to a programmer familiar with block declarations [47] since these declarations support a rich repertoire of conjunctive and disjunction conditions. In fact, a blocking requirement for an n-ary predicate can be expressed in SICStus if and only if it corresponds to a monotonic Boolean function over n variables (SICStus only permits blocking conditions to be given that can be specified as conjunctions of disjunctions of argument positions).

Our analysis draws together a number of strands in program analysis and therefore, for clarity, we summarise its contributions:

- The analysis performs goal-independent suspension analysis, inferring a class of goals that do not lead to a suspension state. As well as having the benefit of generality, this approach leads to a point-and-click mode of operation in which, after directing the analyser at a file, the user merely needs to inspect the results generated by the analysis.
- The analysis exploits a property of *Mon* and *Pos* that enables an exponential number of sub-goal reorderings to be considered in a linear number of logical operations. This result underpins the scalability of the analysis.
- The analysis reduces to two simple bottom-up fixpoint computations – a lfp and a gfp – which makes it surprisingly simple to implement.
- The analysis strikes a good balance between tractability and precision. It is fast enough be frequently used within program development and is precise enough to locate subtle bugs in real programs.

The paper is structured as follows: Section 2 gives a detailed but accessible account of how to apply the analysis. The focus of this section is a single worked example. Section 3 explains the pitfalls of goal-independent suspension analysis; this section gives insight into the design of the analysis. The following sections are more rigorous and incrementally build towards a correctness argument. Section 4 introduces the necessary preliminaries so that the paper is self-contained. Section 5 explains the rôle of Boolean functions in analysis. Section 6 details the analysis itself. Section 7 outlines the implementation and section 8 presents an experimental evaluation. Section 9 reviews related work and Section 10 concludes. This paper is an extended and revised version of [22]. For reasons on continuity, all proofs are relegated to appendix A (which will be available as an electronic appendix to the final version of the paper).

2 Worked example

Consider the Prolog program that is listed below:

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

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

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 append(-, ?, -)` delays (blocks) `append` goals until their arguments are sufficiently instantiated. The dashes in the first and third argument positions specify that a call to `append` is to be delayed until either its first or third argument are bound to non-variable terms. Thus `append` goals can be executed in one of two modes. The analysis 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. This problem can be solved with backward analysis. The backward analysis infers conditions on the input which are sufficient for the query to be solved, without incurring a suspension, with a *local* selection rule [53]. 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. Any program that can be shown to be suspension-free under local selection is suspension-free with a more general selection rule (though the converse does not follow). This worked example is intended to show that local selection fits elegantly with backward reasoning. The analysis itself reduces to three steps: a program abstraction step; least fixpoint (lfp) and a greatest fixpoint (gfp) computation. Subsections 2.1, 2.2 and 2.3 are devoted to each of these steps.

2.1 Program abstraction

Abstraction in turn reduces to two transformations: normalisation and abstraction. Former transforms the Prolog program into a form in which the head and body atoms are flat, that is, the arguments of these atoms are vectors of distinct variables. A normalised version of the `inorder` program in given below:

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

```
:- block append(-, ?, -).
append(L, Ys, A) :- L = [], A = Ys.
append(L, Ys, A) :- L = [X|Xs], A = [X|Zs], append(Xs,Ys,Zs).
```

Body unifications such as `T = tree(L, V, R)` and `A = [V|RI]` make explicit unifications that would otherwise occur during argument passing. To simplify the presentation, the body atoms of an normalised clause are reordered so that unifications precede the remaining atoms. This does not change the suspension behaviour of the program. Moreover, the suspension behaviour of the program is not altered by any new unifications that appear in the normalised program since they will not instantiate any arguments that were uninstantiated in the original program. The result of the latter transform, abstraction, is given below:

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

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

In the abstract version of the program, each unification is replaced with a Boolean function in the normalised program which captures its instantiation dependencies. For example, the unification `A = [V|RI]` is replaced (abstracted) by the Boolean function $\mathtt{A} \leftrightarrow (\mathtt{V} \wedge \mathtt{RI})$. The Boolean function states that `A` will be bound to a ground term during execution if and only if both `V` and `RI` are bound to ground terms during execution. This function is an exemplar of the *Pos* domain, that is, the set of Boolean functions $f : \{\text{true}, \text{false}\}^n \rightarrow \{\text{true}, \text{false}\}$ such that $f(\text{true}, \ldots, \text{true}) = \text{true}$. Other functions such as $\neg x$ (that have little value in expressing groundness dependencies [39]) fall outside *Pos* since $\neg\text{true} = \text{false}$.

Observe, that once the Herbrand constraint `A = [V|RI]` is satisfied, then the function $\mathtt{A} \leftrightarrow (\mathtt{V} \wedge \mathtt{RI})$ describes a property of that state that necessarily holds. The converse is true for the way blocks are abstracted; blocks are abstracted by Boolean functions that describe sufficient conditions for the blocks to hold. For example, the declaration `:- block append(-, ?, -)` is abstracted with an assertion by pairing the function $\mathtt{L} \vee \mathtt{A}$ with the atom `append(L, Ys, A)`. When the function holds, that is, when either `L` or `A` are ground, then the first or second arguments of the call `append(L, Ys, A)` are bound to non-variables terms. Hence the call will not block. The resulting abstract program is used as input to the lfp and gfp.

2.2 Least fixpoint calculation

The second step of the analysis approximates the success patterns of the normalised program (and hence the original program) by computing a lfp of the abstract program. A success pattern is an atom with distinct variables for arguments paired with a formula over those variables. A success pattern summarises the behaviour of an atom by describing the bindings it can make. The lfp of the abstract program can be computed T_P-style [23] 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:

$$F = \left\{ \begin{array}{c} \langle \texttt{inorder}(x_1, x_2), x_1 \leftrightarrow x_2 \rangle \\ \langle \texttt{append}(x_1, x_2, x_3), (x_1 \wedge x_2) \leftrightarrow x_3 \rangle \end{array} \right\}$$

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

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, yet it describes a set of calls that do not suspend under local selection. Iteration commences with D_0 and the call pattern formulae are incrementally strengthened until they are safe, that is, they describe queries which are guaranteed not to suspend. This leads to the following sequence of D_i iterates:

$$D_0 = \left\{ \begin{array}{c} \langle \texttt{inorder}(x_1, x_2), \mathrm{true} \rangle \\ \langle \texttt{append}(x_1, x_2, x_3), \mathrm{true} \rangle \end{array} \right\} \quad D_1 = \left\{ \begin{array}{c} \langle \texttt{inorder}(x_1, x_2), \mathrm{true} \rangle \\ \langle \texttt{append}(x_1, x_2, x_3), x_1 \vee x_3 \rangle \end{array} \right\}$$

$$D_2 = \left\{ \begin{array}{c} \langle \texttt{inorder}(x_1, x_2), x_1 \vee x_2 \rangle \\ \langle \texttt{append}(x_1, x_2, x_3), x_1 \vee x_3 \rangle \end{array} \right\} \; D_3 = D_2$$

The stable iterate D_2 corresponds to the gfp and constitues the result of the analysis. 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, 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 `append(LI, A, I)` whereas if the second argument is ground, then the reverse ordering is sufficient for non-suspension.

The iterate D_{i+1} is computed by putting $D_{i+1} = D_i$ and then revising D_{i+1} by considering each clause $p(\vec{x})$:- $f_1, \ldots, f_m, p_1(\vec{x}_1), \ldots, p_n(\vec{x}_n)$ 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. Recall that a monotonic function is a formula such as $x \vee (y \wedge z)$ and $w \vee (x \wedge (y \vee z))$. More exactly, a monotonic formula over set of variables X is any formula of the form $\vee_{i=1}^{n}(\wedge Y_i)$ where $Y_i \subseteq X$ [14]. Observe that each abstract clause arises from a clause $p(\vec{x})$:- b in the normalised program with a body that is a compound goal of the form $b = e_1, \ldots, e_m, p_1(\vec{x}_1), \ldots, p_n(\vec{x}_n)$ where each e_i is a syntactic equation (unification).

Let d_j denote a monotonic formula that describes the call pattern requirement for $p_j(\vec{x}_j)$ in D_i and let g_j denote the success pattern formula for $p_j(\vec{x}_j)$ in the lfp (that is not necessarily monotonic). A new call pattern for $p(\vec{x})$ is computed using a 6 step algorithm which is given below. To illustrate each step, the presentation will focus on the second clause of `inorder`, and illustrate how the call pattern for `inorder` is updated in D_1 to obtain a new (stronger) requirement for non-suspension in D_2. The explanation will not major on how D_1 is generated because this is not so insightful; the call patterns appearing in the D_1 iterate merely correspond to groundness conditions that satisfy the block declarations of the predicates. The call to `append` will not (immediately) block if its first or third arguments are ground; a call to `inorder` can never (immediately) block because a delay declaration is not specified for this predicate. The 6 steps of the algorithm procede as follows:

- Calculate $g = (\wedge_{i=1}^{m} f_i) \wedge (\wedge_{j=1}^{n} d_j \to g_j)$ that describes the grounding behaviour of the compound goal b. The action of each sub-goal $p_j(\vec{x}_j)$ on the program state is captured by the formula $d_j \to g_j$. The intuition is that if the input groundness requirements d_j are satisfied, then $p_j(\vec{x}_j)$ can be executed without suspension, hence g_j must describe the resulting state. The function $\wedge_{j=1}^{n} d_j \to g_j$ describes the cumulative effect of the sub-goals $p_1(\vec{x}_1), \ldots, p_n(\vec{x}_n)$ on the state. Moreover, each unification e_i never suspends and results in a state that is described by f_i. The formula $\wedge_{i=1}^{m} f_i$ thus expresses the effect of the sub-goals $e_1, \ldots, e_m$. In the case of the second clause of `inorder`, $b = \mathtt{T} \leftrightarrow (\mathtt{L} \wedge \mathtt{V} \wedge \mathtt{R})$, $\mathtt{A} \leftrightarrow (\mathtt{V} \wedge \mathtt{RI})$, $\mathtt{append}(\mathtt{LI}, \mathtt{A}, \mathtt{I})$, $\mathtt{inorder}(\mathtt{L}, \mathtt{LI})$, $\mathtt{inorder}(\mathtt{R}, \mathtt{RI})$ and thus $f_1 = \mathtt{T} \leftrightarrow (\mathtt{L} \wedge \mathtt{V} \wedge \mathtt{R})$ and $f_2 = \mathtt{A} \leftrightarrow (\mathtt{V} \wedge \mathtt{RI})$. Moreover from D_0 it follows that $d_1 = \mathtt{I} \vee \mathtt{LI}$, $d_2 = \text{true}$ and $d_3 = \text{true}$ and from the lfp, $g_1 = (\mathtt{LI} \wedge \mathtt{A}) \leftrightarrow \mathtt{I}$, $g_2 = \mathtt{L} \leftrightarrow \mathtt{LI}$ and $g_3 = \mathtt{R} \leftrightarrow \mathtt{RI}$.

The function g that results from this step is given below (even the simplest examples give rise to complicated formulae, illustrating the need for automation):

$$g = \begin{cases} (\mathtt{T} \wedge \mathtt{I} \wedge \mathtt{V} \wedge \mathtt{A} \wedge \mathtt{R} \wedge \mathtt{RI} \wedge \mathtt{L} \wedge \mathtt{LI}) \vee \\ (\neg\mathtt{T} \wedge \neg\mathtt{I} \wedge \mathtt{V} \wedge \mathtt{A} \wedge \mathtt{R} \wedge \mathtt{RI} \wedge \neg\mathtt{L} \wedge \neg\mathtt{LI}) \vee \\ (\neg\mathtt{T} \wedge \neg\mathtt{I} \wedge \mathtt{V} \wedge \neg\mathtt{A} \wedge \neg\mathtt{R} \wedge \neg\mathtt{RI} \wedge \mathtt{L} \wedge \mathtt{LI}) \vee \\ (\neg\mathtt{T} \wedge \neg\mathtt{I} \wedge \mathtt{V} \wedge \neg\mathtt{A} \wedge \neg\mathtt{R} \wedge \neg\mathtt{RI} \wedge \neg\mathtt{L} \wedge \neg\mathtt{LI}) \vee \\ (\neg\mathtt{T} \wedge \neg\mathtt{I} \wedge \neg\mathtt{V} \wedge \neg\mathtt{A} \wedge \mathtt{R} \wedge \mathtt{RI} \wedge \mathtt{L} \wedge \mathtt{LI}) \vee \\ (\neg\mathtt{T} \wedge \neg\mathtt{I} \wedge \neg\mathtt{V} \wedge \neg\mathtt{A} \wedge \mathtt{R} \wedge \mathtt{RI} \wedge \neg\mathtt{L} \wedge \neg\mathtt{LI}) \vee \\ (\neg\mathtt{T} \wedge \neg\mathtt{I} \wedge \neg\mathtt{V} \wedge \neg\mathtt{A} \wedge \neg\mathtt{R} \wedge \neg\mathtt{RI} \wedge \mathtt{L} \wedge \mathtt{LI}) \vee \\ (\neg\mathtt{T} \wedge \neg\mathtt{I} \wedge \neg\mathtt{V} \wedge \neg\mathtt{A} \wedge \neg\mathtt{R} \wedge \neg\mathtt{RI} \wedge \neg\mathtt{L} \wedge \neg\mathtt{LI}) \end{cases}$$

- Compute $d = \wedge_{j=1}^{n} d_j$ which describes a groundness property sufficient for executing all the sub-goals $p_j(\vec{x}_j)$ in b without suspension. For the `inorder` clause, this results in the Boolean function:

$$d = \mathtt{I} \vee \mathtt{LI}$$

- Calculate $g \to d$ which is a function that describes a grounding property which, if satisfied by the state when b is called, ensures that b can be scheduled under local selection without suspension. (This tactic relies on the d_j being monotonic and is justified in section 5.3.) The function $g \to d$ is given below:

$$g \to d = \begin{cases} \mathtt{T} \vee \\ \mathtt{I} \vee \\ (\mathtt{V} \wedge \mathtt{A} \wedge \mathtt{R} \wedge \mathtt{RI} \wedge \mathtt{L}) \vee \\ (\mathtt{V} \wedge \mathtt{A} \wedge \mathtt{R} \wedge \mathtt{RI} \wedge \neg\mathtt{L} \wedge \mathtt{LI}) \vee \\ (\mathtt{V} \wedge \mathtt{A} \wedge \mathtt{R} \wedge \neg\mathtt{RI}) \vee \\ (\mathtt{V} \wedge \mathtt{A} \wedge \neg\mathtt{R}) \vee \\ (\mathtt{V} \wedge \neg\mathtt{A} \wedge \mathtt{R}) \vee \\ (\mathtt{V} \wedge \neg\mathtt{A} \wedge \neg\mathtt{R} \wedge \mathtt{RI}) \vee \\ (\mathtt{V} \wedge \neg\mathtt{A} \wedge \neg\mathtt{R} \wedge \neg\mathtt{RI} \wedge \mathtt{L}) \vee \\ (\mathtt{V} \wedge \neg\mathtt{A} \wedge \neg\mathtt{R} \wedge \neg\mathtt{RI} \wedge \neg\mathtt{L} \wedge \mathtt{LI}) \vee \\ (\neg\mathtt{V} \wedge \mathtt{A}) \vee \\ (\neg\mathtt{V} \wedge \neg\mathtt{A} \wedge \mathtt{R} \wedge \mathtt{RI} \wedge \mathtt{L}) \vee \\ (\neg\mathtt{V} \wedge \neg\mathtt{A} \wedge \mathtt{R} \wedge \mathtt{RI} \wedge \neg\mathtt{L} \wedge \mathtt{LI}) \vee \\ (\neg\mathtt{V} \wedge \neg\mathtt{A} \wedge \mathtt{R} \wedge \neg\mathtt{RI}) \vee \\ (\neg\mathtt{V} \wedge \neg\mathtt{A} \wedge \neg\mathtt{R} \wedge \mathtt{RI}) \vee \\ (\neg\mathtt{V} \wedge \neg\mathtt{A} \wedge \neg\mathtt{R} \wedge \neg\mathtt{RI} \wedge \mathtt{L}) \vee \\ (\neg\mathtt{V} \wedge \neg\mathtt{A} \wedge \neg\mathtt{R} \wedge \neg\mathtt{RI} \wedge \neg\mathtt{L} \wedge \mathtt{LI}) \end{cases}$$

Note that this function already expresses interesting non-suspension properties: among aother things, it states that if the compound goal b is called with the variable `T` ground, then b will not suspend providing that d_1, d_2 and d_3 are genuine criteria for non-suspension of the sub-goals. Indeed, if b is called with a ground `T`, then the unification `T = tree(L, V, R)` will ground `L` and `R`. The sub-goals `inorder(L, LI)` and `inorder(R, RI)` will then bind `LI` and `RI` to ground terms. The property d_1 asserts that `append(LI, A, L)` will not suspend if called with `LI` or `L` ground; this is satisfied, hence `T` is sufficient for non-suspension of b providing d_1, d_2 and d_3 are sufficient for non-suspension of the sub-goals.

- The function $g \to d$ contains variables, such as `V` and `LI`, which do not occur in the head of the clause. Clauses of $g \to d$ which contain these variables, such as $(\mathtt{V} \wedge \mathtt{A} \wedge \mathtt{R} \wedge \mathtt{RI} \wedge \mathtt{L})$, describe non-suspension properties that can never be satisfied by the call $\mathtt{inorder}(\mathtt{T}, \mathtt{I})$. The call can only constrain the variables `T` and `I`; clauses which contain other variables can never be satisfied and therefore can be eliminated to simplify the formula. Elimination is performed by computing $d' = \forall_Y(g \to d)$ where Y is the set of variables not present in the head, $\forall_{\{y_1 \ldots y_n\}}(f) = \forall_{y_1}(\ldots \forall_{y_n}(f))$ and a single variable is eliminated by:

$$\forall_x(f) = \begin{cases} f' & f' \in Pos \\ \text{false} & \text{otherwise} \end{cases} \qquad \text{where} \quad f' = f[x \mapsto \text{false}] \wedge f[x \mapsto \text{true}]$$

In general $\forall_x(f)$ entails f, that is, if $\forall_x(f)$ holds then it follows that f also holds. Thus if $g \to d$ is a prerequisite sufficient for scheduling b under local selection without suspension, then so is d'. The formula d' is computed by successively eliminating variables from $g \to d$ by calculating $\forall_{\mathtt{L}}(g \to d)$, $\forall_{\mathtt{V}}(\forall_{\mathtt{L}}(g \to d))$, $\forall_{\mathtt{R}}\forall_{\mathtt{V}}(\forall_{\mathtt{L}}(g \to d))$, etc. The net result is $d' = \mathtt{T} \vee \mathtt{I}$ and to illustrate the technique, the result of the first elimination step is:

$$\forall_{\mathtt{L}}(g \to d) = \begin{cases} \mathtt{T}\vee \\ \mathtt{I}\vee \\ (\mathtt{V} \wedge \mathtt{A} \wedge \mathtt{R} \wedge \mathtt{RI} \wedge \mathtt{LI})\vee \\ (\mathtt{V} \wedge \mathtt{A} \wedge \mathtt{R} \wedge \neg\mathtt{RI})\vee \\ (\mathtt{V} \wedge \mathtt{A} \wedge \neg\mathtt{R})\vee \\ (\mathtt{V} \wedge \neg\mathtt{A} \wedge \mathtt{R})\vee \\ (\mathtt{V} \wedge \neg\mathtt{A} \wedge \neg\mathtt{R} \wedge \mathtt{RI})\vee \\ (\mathtt{V} \wedge \neg\mathtt{A} \wedge \neg\mathtt{R} \wedge \neg\mathtt{RI} \wedge \mathtt{LI})\vee \\ (\neg\mathtt{V} \wedge \mathtt{A})\vee \\ (\neg\mathtt{V} \wedge \neg\mathtt{A} \wedge \mathtt{R} \wedge \mathtt{RI} \wedge \mathtt{LI})\vee \\ (\neg\mathtt{V} \wedge \neg\mathtt{A} \wedge \mathtt{R} \wedge \neg\mathtt{RI})\vee \\ (\neg\mathtt{V} \wedge \neg\mathtt{A} \wedge \neg\mathtt{R} \wedge \mathtt{RI})\vee \\ (\neg\mathtt{V} \wedge \neg\mathtt{A} \wedge \neg\mathtt{R} \wedge \neg\mathtt{RI} \wedge \mathtt{LI}) \end{cases}$$

- Compute a monotonic function d'' that entails d'. This is required so that in the subsequent steps of the analysis each d_i is monotonic. In the case of `inorder`, the Boolean function $d' = \mathtt{T} \vee \mathtt{I}$ is already monotonic, hence $d'' = d'$ and thus this step is superfluous. However, if $d' = \mathtt{T} \to \mathtt{I}$ say, then it would be necessary to replace d' with $d'' = \mathtt{I}$ – the greatest monotonic function that entails d'. Since d'' entails d', it follows that because d' is a sufficient condition for scheduling b under local selection without suspension, then so is d''.

- Replace the pattern $\langle p(\vec{x}), d'''\rangle$ in D_{i+1} with $\langle p(\vec{x}), d'' \wedge d'''\rangle$, thereby (possibly) strengthening the call pattern for $p(\vec{x})$. For the running example, D_1 is thus updated to contain $\langle \mathtt{inorder}(\mathtt{T}, \mathtt{I}), \mathtt{T} \vee \mathtt{I}\rangle$.

Continuing with this procedure, the gfp is reached and checked in three iterations.

3 The pitfalls of reversing suspension analysis

3.1 Merging non-suspension conditions for a compound goal

The correctness argument that underpins the analysis requires the non-suspension conditions are described as monotonic Boolean functions. It would appear that this requirement could be relaxed by separately considering each sub-goal reordering. For example, the compound goal $p_1(\vec{x}_1)$,

$p_2(\vec{x}_2)$, $p_3(\vec{x}_3)$ could be analysed by merely considering leftmost selection and inferring a non-suspension condition each of the following reorderings:

$$\begin{array}{lll} p_1(\vec{x}_1), p_2(\vec{x}_2), p_3(\vec{x}_3) & p_2(\vec{x}_1), p_1(\vec{x}_1), p_3(\vec{x}_3) & p_3(\vec{x}_3), p_1(\vec{x}_1), p_2(\vec{x}_2) \\ p_1(\vec{x}_1), p_3(\vec{x}_3), p_2(\vec{x}_2) & p_2(\vec{x}_2), p_3(\vec{x}_1), p_1(\vec{x}_1) & p_3(\vec{x}_3), p_2(\vec{x}_2), p_1(\vec{x}_1) \end{array}$$

If $f_1, \ldots, f_6$ described conditions sufficient for non-suspension for these reorderings, then one would think that the function $\vee_{i=1}^{6} f_i$ represented a condition sufficient for the compound goal $p_1(\vec{x}_1), p_2(\vec{x}_2), p_3(\vec{x}_3)$ to be executed without suspension under local selection. However, this is not so.

To illustrate, consider the compound goal `p(X, Y), q(X, Y)`. Depending on the delay criteria specified for the predicates `p` and `q`, under local selection, the goal `p(X, Y)` could be executed before `q(X, Y)` or vice versa. A condition sufficient for non-suspension for the particular scheduling `p` then `q` may differ from the condition sufficient for non-suspension under the ordering `q` then `p`. The problem is therefore how to compute a condition that ensures that at least one scheduling does not suspend. The following logic program illustrates the subtlety of this problem:

```
:- block p(-, ?).
p(X, Y) :- X = f(a, a), Y = f(a, a).

q(X, Y) :- X = f(a, a), r(Y).

:- block r(-).
r(Y) :- Y = f(a, a).
```

The predicate `p` suspends until its first argument is non-variable and then it proceeds to unify both its arguments to the term `f(a, a)`. Observe that if `X` is ground when `p(X, Y)` is invoked, then the call will not suspend. This sufficient condition for non-suspension can be described by the Boolean formula `X`. The formula describes all those substitutions (and therefore all calls) which ground the variable `X`. Now consider the predicate `q`. The body of its clause consists of the unification `X = f(a, a)` and the call `r(Y)`. The predicate `r` suspends until its argument is instantiated and then it unifies the argument with the term `f(a, a)`. The call `q(X, Y)` (and those it invokes) will not suspend if `Y` is ground; this requirement can be described with the Boolean formula `Y`. However, less obviously, the formula $\texttt{X} \rightarrow \texttt{Y}$ describes a strictly larger class of calls each of which ensures non-suspension of `q(X, Y)`. This formula can be interpreted stating the requirement that `Y` must be ground if `X` is ground. As well as including substitutions that ground `Y`, it includes substitutions of the form $\theta_1 = \{\mathtt{X} \mapsto \mathtt{f(Y,Z)}\}$, $\theta_2 = \{\mathtt{X} \mapsto \mathtt{Y}\}$ and $\theta_3 = \{\mathtt{X} \mapsto \mathtt{g(W,Y,Z)}\}$. If `q(X, Y)` is called with either θ_1 or θ_2, then the unification `X = f(a, a)` can be scheduled before the call `r(Y)` grounding `Y` and thereby ensuring that the call `r(Y)` does not suspend. On the other hand, if `q(X, Y)` is called with θ_3, then the unification fails so that non-suspension cannot occur. To summarise, sets of non-suspending calls for `p(X, Y)` and `q(X, Y)` are described by the formulae `X` and $\mathtt{X} \rightarrow \mathtt{Y}$.

Now consider the compound goal `p(X, Y), q(X, Y)`. Under local selection, either `p` is scheduled before `q` or `q` is scheduled before `p`. Observe that in the former case the bindings created by `p` ensure that `q` cannot suspend whereas in the latter case `p` cannot suspend because of the bindings made by `q`. Therefore the formulae `X` and $\mathtt{X} \rightarrow \mathtt{Y}$ represent (different) non-suspension conditions for the compound goal. Since either of these two scheduling scenarios ensure non-suspension, it seems reasonable that calls described by $\mathtt{X} \vee (\mathtt{X} \rightarrow \mathtt{Y})$ will also not suspend. However, $\mathtt{X} \vee (\mathtt{X} \rightarrow \mathtt{Y}) = \mathrm{true}$ and true describes a set which includes the empty substitution ϵ under which the compound goal `p(X, Y), q(X, Y)` suspends. The conjunction $\mathtt{X} \wedge (\mathtt{X} \rightarrow \mathtt{Y}) = (\mathtt{X} \wedge \mathtt{Y})$ is sufficient for non-suspension but this condition is stronger than `X` which is not desirable. The issue, therefore, it how to merge non-suspension conditions that arise from different goal scheduling without compromising correctness yet without enforcing overly strong requirements.

To address this issue, it is important to realise that $\vee$ is an unsatisfactory tactic for merging non-suspension requirements because the class of Boolean function used within the analysis, *Pos* [1], is not closed under disjunctive completion [18]. It is not closed under disjunction completion since for certain Boolean functions $f_1, f_2 \in Pos$ the set of substitutions described by $f_1 \vee f_2$ is strictly larger than the union of those sets of substitutions described by f_1 and f_2 separately. The solution is not to adopt a different merge operator but apply $\vee$ within a sub-domain of *Pos* that is closed under disjunctive completion. The most expressive sub-domain of *Pos* which possesses this property is *Mon*. Two non-suspension requirements f_1 and f_2 can be merged by computing *Mon* formulae g_1 and g_2 which are strong enough to ensure f_1 and f_2 hold respectively. The *Mon* formula $g_1 \vee g_2$ is then a sufficient condition for non-suspension. To continue with the above example, if $f_1 = \mathtt{X}$ and $f_2 = \mathtt{X} \rightarrow \mathtt{Y}$, then $g_1 = \mathtt{X}$ and $g_2 = \mathtt{Y}$, hence $g_1 \vee g_2 = \mathtt{X} \vee \mathtt{Y}$ which indeed is sufficient for ensuring that the compound goal `p(X, Y), q(X, Y)` does not suspend.

3.2 Modelling the synchronisation within a compound goal

One novelty of the analysis is its use of a Boolean function $g \rightarrow d$ to infer a requirement that is sufficient for non-suspension: if $g \rightarrow d$ describes the state when a compound goal b is called, then b can be scheduled under local selection without suspension. The function $g \rightarrow d$ is constructed from g and d which are themselves defined in terms of the g_i and d_i – success patterns and (monotonic) non-suspension conditions – of the atomic sub-goals of b. One is tempted to relax the requirement that the d_i are monotonic since this would infer richer and therefore more insightful non-suspension conditions. However, this can lead to disaster.

To illustrate the problem, consider again the compound goal `p(X, Y)`, `q(X, Y)` and recall that the individual sub-goals `p(X, Y)` and `q(X, Y)` will not suspend when called with inputs described by $\mathtt{X}$ and $\mathtt{X} \rightarrow \mathtt{Y}$ respectively. Observe too these non-suspending calls to `p` and `q` will both ground `X` and `Y`, hence $\mathtt{X} \wedge \mathtt{Y}$ describes their output grounding behaviour. Therefore, the input-output behaviour of the compound goal `p(X, Y), q(X, Y)` is described by:

$$g = (\mathtt{X} \rightarrow (\mathtt{X} \wedge \mathtt{Y})) \wedge ((\mathtt{X} \rightarrow \mathtt{Y}) \rightarrow (\mathtt{X} \wedge \mathtt{Y}))$$

The formula e can be read as stating that for input bindings which ground `X`, then the compound goal will generate output bindings that ground `Y`. Moreover, for any input bindings described by $\mathtt{X} \rightarrow \mathtt{Y}$ then the compound goal will generate output bindings which ensure that the groundness property $\mathtt{X} \wedge \mathtt{Y}$ holds. However,

$$g = (\mathtt{X} \rightarrow \mathtt{Y}) \wedge ((\mathtt{X} \rightarrow \mathtt{Y}) \rightarrow (\mathtt{X} \wedge \mathtt{Y})) = (\mathtt{X} \rightarrow \mathtt{Y}) \wedge (\mathtt{X} \wedge \mathtt{Y}) = \mathtt{X} \wedge \mathtt{Y}$$

The deficiency in the model becomes apparent when the formula $g \rightarrow d$ is calculated to infer a class of bindings for which the compound goal can be scheduled in some non-suspending fashion. Recall that the formula d represents a grounding property sufficient for scheduling both `p(X, Y)` and `q(X, Y)` without suspension:

$$d = \mathtt{X} \wedge (\mathtt{X} \rightarrow \mathtt{Y}) = \mathtt{X} \wedge \mathtt{Y}$$

Therefore

$$g \rightarrow d = (\mathtt{X} \wedge \mathtt{Y}) \rightarrow (\mathtt{X} \wedge \mathtt{Y}) = \mathrm{true}$$

All bindings (including the empty substitution) are described by the formula true, yet every local scheduling of `p(X, Y), q(X, Y)` induces a suspension when the goal is called with X and Y unbound. If the sub-goal `p(X, Y)` is selected first, then it will suspend on X; if `q(X, Y)` is selected first, then its sub-goal `r(Y)` will suspend on Y. Note in this case, sub-goals can be interleaved to avoid suspension, but this is prohibited under local selection. Thus the method, as applied above, cannot correctly infer non-suspension conditions sufficient for local selection.

The problem stems from g and that $(\mathtt{X} \rightarrow (\mathtt{X} \wedge \mathtt{Y})) = \mathtt{X} \rightarrow \mathtt{Y}$ matches the antecedent of the formula $(\mathtt{X} \rightarrow \mathtt{Y}) \rightarrow (\mathtt{X} \wedge \mathtt{Y})$. This problem can be avoided by replacing the non-suspension conditions for `p(X, Y)` and `q(X, Y)`, namely X and $\mathtt{X} \rightarrow \mathtt{Y}$, with stronger conditions that are monotonic. The functions X and Y are both monotonic and $\mathtt{X} \rightarrow \mathtt{Y}$ holds whenever Y holds. Reworking g, d and $g \rightarrow d$ with these monotonic non-suspension conditions gives:

$$g = (\mathtt{X} \rightarrow (\mathtt{X} \wedge \mathtt{Y})) \wedge (\mathtt{Y} \rightarrow (\mathtt{X} \wedge \mathtt{Y})) = (\mathtt{X} \rightarrow \mathtt{Y}) \wedge (\mathtt{Y} \rightarrow \mathtt{X}) = (\mathtt{Y} \leftrightarrow \mathtt{X})$$

$$d = (\mathtt{X} \wedge \mathtt{Y}) \qquad\qquad g \rightarrow d = (\mathtt{X} \leftrightarrow \mathtt{Y}) \rightarrow (\mathtt{X} \wedge \mathtt{Y}) = \mathtt{X} \vee \mathtt{Y}$$

which yields the condition $\mathtt{X} \vee \mathtt{Y}$ that is sufficient for ensuring non-suspension of the compound goal `p(X, Y), q(X, Y)` even when it is executed under local selection. Correctness is not compromised in this formulation because $\mathtt{X} \rightarrow \mathtt{Y}$ does not match the antecedent of $\mathtt{Y} \rightarrow \mathtt{X}$, that is, Y; neither is it stronger than Y. This is because $\mathtt{X} \rightarrow \mathtt{Y}$ is satisfied by the truth assignment $\{\mathtt{X} \mapsto \text{false}, \mathtt{Y} \mapsto \text{false}\}$ yet this assignment can never satisfy any monotonic function (other than true).

4 Logic programming preliminaries

This section servers as a self-contained reference library for the sequel of the paper; it contains the formalism required to argue correctness of the analysis. The presentation is dense since much of this material should be familiar. The casual reader can skim this section with the exception of subsections 4.3–4.5.

4.1 Sequences and vectors

Let ϵ denote the empty sequence, let . denote concatenation, let $\|s\|$ denote the length of a sequence s, and let S^* denote the set of sequences whose elements are drawn from a set S.

4.2 Terms, equations, substitutions and unifiers

Let $Term$ denote the set of (possibly infinite) terms over an alphabet of functor symbols $Func$ and a (denumerable) universe of variables Var where $Func \cap Var = \emptyset$. Let $var(t)$ denote the set of variables occurring in the term t. An equation e is a pair $(s = t)$ where $s, t \in Term$. A finite set of equations is denoted E and Eqn denotes the set of finite sets of equations.

A substitution is a map $\theta : Var \rightarrow Term$ where $dom(\theta) = \{u \in Var \mid \theta(u) \neq u\}$ is finite. Let $rng(\theta) = \cup\{var(\theta(u)) \mid u \in dom(\theta)\}$ and let $var(\theta) = dom(\theta) \cup rng(\theta)$. A substitution θ is idempotent iff $\theta \circ \theta = \theta$, or equivalently, iff $dom(\theta) \cap rng(\theta) = \emptyset$. 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 $x \in dom(\theta)$ in t with $\theta(x)$. Also define $\theta(E) = \{\theta(s) = \theta(t) \mid (s = t) \in E\}$. The map $\mathrm{eqn} : Sub \rightarrow Eqn$ is defined $\mathrm{eqn}(\theta) = \{x = \theta(x) \mid x \in dom(\theta)\}$. Composition $\theta \circ \psi$ of two substitutions is defined so that $(\theta \circ \psi)(u) = \theta(\psi(u))$ for all $u \in Var$. Composition induces the (more general than) relation $\leq$ defined by $\theta \leq \psi$ iff there exists $\delta \in Sub$ 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).

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 $mgu(E) \neq \emptyset$ [35].

4.3 Syntax of 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. A sequence of equations $e_1, \ldots, e_m$ where each $e_i = (s_i = t_i)$ is equivalent to the single equation $f(s_1, \ldots, s_m) = f(t_1, \ldots, t_m)$ where f is any functor $f \in Func$. Thus it is sufficient to suppose that a logic program P is a finite set of clauses $p(\vec{x})$:- e, g where the body e, g is composed of a single equation e and a sequence (or equivalently a multiset) of atoms $g \in Goal$ where $Goal = Atom^*$.

Dynamic scheduling assertions (block declarations) are formalised as a predicate $select \subseteq Atom \times Sub$ that satisfies the following conditions:

- if $select(p(\vec{x}), \theta)$ holds then $select(p(\vec{x}), \delta \circ \theta)$ holds for all $\delta \in Sub$;

- if $select(p(\vec{x}), \theta)$ holds then $select(p(\vec{y}), \theta \circ \rho)$ holds where $\vec{x} = \langle x_1, \ldots, x_n \rangle$, $\vec{y} = \langle y_1, \ldots, y_n \rangle$ and $\rho = \{y_1 \mapsto x_1, \ldots, y_n \mapsto x_n\}$.

Example 4.1 Consider the declaration `:- block append(-, ?, -)`. Then $select(\mathtt{append}(x_1, x_2, x_3), \theta)$ holds iff $\theta(x_1) \notin Var$ or $\theta(x_3) \notin Var$.

4.4 Fixpoint semantics of logic programs

A fixpoint semantics of P is defined in terms of an immediate consequences operator $\mathcal{F}$ that operators on a complete lattice. To construct this lattice, let $Base = \{\langle p(\vec{x}), \theta \rangle \mid p(\vec{x}) \in Atom \wedge \theta \in Sub\}$. The more general than relation induces a pre-order on $Base$ by $\langle p(\vec{y}), \psi \rangle \leq \langle p(\vec{x}), \theta \rangle$ iff $\theta(\vec{x}) \leq \psi(\vec{y})$ (note the order reversal). The closure of a set $I \subseteq Base$ is defined by $down(I) = \{a \in Base \mid \exists b \in I . a \leq b\}$. The set of closed sets is Int $= \{I \subseteq Base \mid down(I) = I\}$. Then $\langle Int, \subseteq, \cup, \cap, Base, \emptyset \rangle$ is a complete lattice, in fact, Int constitutes a domain of interpretations.

Definition 4.2 Given a logic program P, the operator $\mathcal{F} : Int \rightarrow Int$ is defined:

$$\mathcal{F}(I) = \left\{ \langle p(\vec{x}), \theta \rangle \;\middle|\; \begin{array}{l} p(\vec{x}) \text{ :- } e, p_1(\vec{x}_1), \ldots, p_n(\vec{x}_n) \in P \\ \theta \in unify(\{e\}) \wedge \langle p_i(\vec{x}_i), \theta \rangle \in I \end{array} \right\}$$

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

4.5 Operational semantics of logic programs

The operational semantics for dynamic scheduling is defined in terms of transition systems between states where $State = (Goal \times Sub) \cup \{susp\}$ where $susp$ indicates a suspension state. Transitions are annotated with an integer k that records the number of resolution steps that can be performed before a suspension arises.

Definition 4.3 (transition system for dynamic scheduling) Given a program P, $\twoheadrightarrow_{delay} \subseteq State \times State$ is the least relation such that:

$$\frac{s_1 {\twoheadrightarrow_{delay}}^{k_1} s_2 \qquad s_2 {\twoheadrightarrow_{delay}}^{k_2} s_3}{s_1 {\twoheadrightarrow_{delay}}^{k_1 + k_2} s_3} \qquad \frac{\forall i \in [1, n] . \neg select(p_i(\vec{x}_i), \theta)}{\langle \{p_i(\vec{x}_i) \mid i \in [1, n]\}, \theta \rangle {\twoheadrightarrow_{delay}}^{0} susp}$$

$$\frac{\begin{array}{c} select(p(\vec{x}),\theta) \\ c = p(\vec{y}) \text{ :- } e, b \in \rho(P) \text{ where } \rho \in Ren \\ \mathrm{var}(c) \cap \mathrm{var}(\langle \{p(\vec{x})\} \cup g, \theta \rangle) = \emptyset \\ \delta \in imgu(\{\theta(\vec{x}) = \vec{y}\} \cup e) \end{array}}{\langle \{p(\vec{x})\} \cup g, \theta \rangle \twoheadrightarrow_{delay}{}^{1} \langle b \cup g, \delta \circ \theta \rangle}$$

The following transition system additionally enforces the requirement that the selection rule is local:

Definition 4.4 (transition system for local selection)
Given a program P, $\twoheadrightarrow_{local} \subseteq State \times State$ is the least relation such that:

$$\frac{s_1 \twoheadrightarrow_{local}{}^{k_1} s_2 \qquad s_2 \twoheadrightarrow_{local}{}^{k_2} s_3}{s_1 \twoheadrightarrow_{local}{}^{k_1+k_2} s_3} \qquad \frac{\langle \{p(\vec{x})\}, \theta \rangle \twoheadrightarrow_{local}{}^{k} \langle \epsilon, \psi \rangle}{\langle \{p(\vec{x})\} \cup g, \theta \rangle \twoheadrightarrow_{local}{}^{k} \langle g, \psi \rangle}$$

$$\frac{\neg select(p(\vec{x}),\theta)}{\langle \{p(\vec{x})\}, \theta \rangle \twoheadrightarrow_{local}{}^{0} susp} \qquad \frac{\forall i \in [1,n]. \langle \{p_i(\vec{x}_i)\}, \theta \rangle \twoheadrightarrow_{local}{}^{k_i} susp}{\langle \{p_i(\vec{x}_i) \mid i \in [1,n]\}, \theta \rangle \twoheadrightarrow_{local}{}^{\max\{k_i \mid i \in [1,n]\}} susp}$$

$$\frac{\begin{array}{c} select(p(\vec{x}),\theta) \\ c = p(\vec{y}) \text{ :- } e, b \in \rho(P) \text{ where } \rho \in Ren \\ \mathrm{var}(c) \cap \mathrm{var}(\langle \{p(\vec{x})\}, \theta \rangle) = \emptyset \\ \delta \in imgu(\{\theta(\vec{x}) = \vec{y}\} \cup e) \end{array}}{\langle \{p(\vec{x})\}, \theta \rangle \twoheadrightarrow_{local}{}^{1} \langle b, \delta \circ \theta \rangle}$$

Definition 4.5 The operational semantics induced by $\twoheadrightarrow_{delay}$ is the transitive closure of $\twoheadrightarrow_{delay}$ applied to the set of (atomic) goals, that is,

$$\mathcal{O}_{delay}(P) = \{ \langle p(\vec{x}), \psi \rangle \mid \exists \langle p(\vec{x}), \theta \rangle \in State \;.\; \langle p(\vec{x}), \theta \rangle \twoheadrightarrow_{delay}{}^{\star} \langle \epsilon, \psi \rangle \}$$

$\mathcal{O}_{local}(P)$ is defined analogously in terms of $\twoheadrightarrow_{local}$.

The relationship between the two operational semantics and the fixpoint semantics is stated in the following results:

Proposition 4.6 If $\langle g, \theta \rangle \twoheadrightarrow_{delay}{}^{l} susp$ then $\langle g, \theta \rangle \twoheadrightarrow_{local}{}^{k} susp$ where $k \leq l$.

Theorem 4.7 $down(\mathcal{O}_{local}(P)) \subseteq down(\mathcal{O}_{delay}(P)) \subseteq \mathcal{F}(P)$.

5 Boolean functions

This section reviews Boolean functions before moving to introduce new properties of Boolean functions that are particularly pertinent to suspension inference. A Boolean function is a function $f : Bool^n \to Bool$ where $n \geq 0$ and $Bool = \{\text{true}, \text{false}\}$. A Boolean function can be represented by a propositional formula over $X \subseteq Var$ where $|X| = n$ in fact, henceforth, X is finite. 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 true is also followed.

Definition 5.1 The map $model_X : Bool_X \to \wp(\wp(X))$ is defined by: $model_X(f) = \{M \subseteq X \mid (\wedge_{x \in M} x) \wedge (\wedge_{y \in X \setminus M} \neg y) \models f\}$.

Example 5.2 *If* $X = \{x, y\}$*, then the Boolean function* $\{\langle \text{true}, \text{true} \rangle \mapsto \text{true}, \langle \text{true}, \text{false} \rangle \mapsto \text{false}, \langle \text{false}, \text{true} \rangle \mapsto \text{false}, \langle \text{false}, \text{false} \rangle \mapsto \text{false}\}$ *can be represented by the formula* $x \wedge y$*. Moreover,* $model_X(x \wedge y) = \{\{x, y\}\}$*,* $model_X(x \vee y) = \{\{x\}, \{y\}, \{x, y\}\}$*,* $model_X(\text{false}) = \emptyset$ *and* $model_X(\text{true}) = \wp(\wp(X)) = \{\emptyset, \{x\}, \{y\}, \{x, y\}\}$*.*

5.1 Classes of Boolean functions

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

Definition 5.3 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 false); Def_X denote the set of positive functions over X that are definite (augmented with false); and Mon_X denote the set of monotonic Boolean functions over X (that includes false). 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 $f \neq \text{false}$, then $f = \wedge_{i=1}^{m}(y_i \leftarrow \wedge Y_i)$ for some $y_i \in X$ and $Y_i \subseteq X$ [14]. Moreover, if $f \in Mon_X$ and $f \neq \text{false}$, then $f = \vee_{i=1}^{m}(\wedge Y_i)$ where $Y_i \subseteq X$ [11, Proposition 2.1]. The 4-tuple $\langle Pos_X, \models, \wedge, \vee \rangle$ is a finite lattice and $\langle Mon_X, \models, \wedge, \vee \rangle$ is a sub-lattice (whereas Def_X is not a sub-lattice as witnessed by the join of x and y in figure 1).

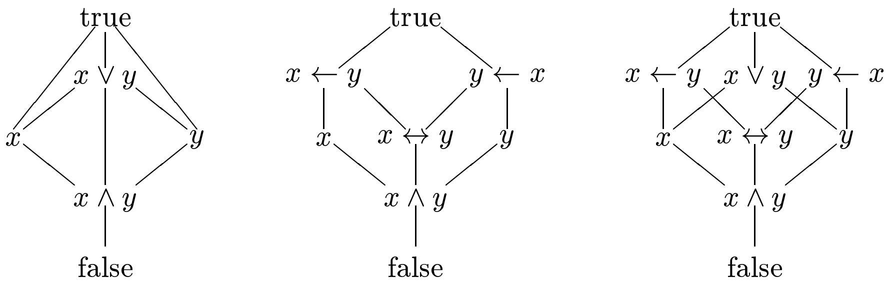

Figure 1: Hasse diagrams for Mon_X, Def_X and Pos_X for the dyadic case $X = \{x, y\}$

5.2 Projection of monotonic and positive Boolean functions

Existential quantification for Pos_X is defined by Schröder elimination, that is, $\exists_x f = f[x \mapsto \text{true}] \vee f[x \mapsto \text{false}]$. If $f \in Pos_X$ then not only $\exists_x f \in Pos_X$ [1] but $\exists_x f = \wedge\{g \in Pos_{X \setminus \{x\}} \,|\, f \models g\}$. Thus there is no positive function g over $X \setminus \{x\}$ such that $f \models g$ which fewer models than $\exists_x f$. Universal projection is defined $\forall_x f = f'$ if $f' \in Pos_X$ otherwise $\forall_x f = \text{false}$ where $f' = f[x \mapsto \text{false}] \wedge f[x \mapsto \text{true}]$. Not only $\forall_x f \in Pos_X$ if $f \in Pos_X$ and $\forall_x f \in Mon_X$ if $f \in Mon_X$ but analogous to existential projection, the following properties hold for universal projection over the domains Pos_X and Mon_X.

Proposition 5.4 *If $f \in Pos_X$ then $\forall_x f = \vee\{g \in Pos_{X \setminus \{x\}} \,|\, g \models f\}$*

Corollary 5.5 *If $f \in Mon_X$ then $\forall_x f = \vee\{g \in Mon_{X \setminus \{x\}} \,|\, g \models f\}$*

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 $\overline{\exists}_Y f = \exists_{X \setminus Y} f$ and $\overline{\forall}_Y f = \forall_{X \setminus Y} f$.

5.3 Reordering property involving monotonic Boolean functions

One key idea behind the analysis is to compute a Boolean function of the form $(\wedge_{i=1}^n d_i \rightarrow g_i) \rightarrow (\wedge_{i=1}^n d_i)$ to characterise those states under which a compound goal $p_1(\vec{x}_1), \ldots, p_n(\vec{x}_n)$ can be executed without incurring a suspension. This tactic is founded on the following proposition; the accompanying example illustrates how the result can be applied.

Proposition 5.6 *Let $f \in Def_X$, $f_i \in Pos_X$, $d_i \in Mon_X$ and $f \not\models d_i$ for all $i \in [1, m]$. Then $f \wedge \wedge_{j=1}^{m}(d_j \rightarrow f_j) \not\models d_i$ for all $i \in [1, m]$.*

Example 5.7 Consider the following program where the block declaration for `q` ensures that a call to `q` will suspend until both its arguments are instantiated. Observe that the compound goal `p(X,Z), q(X,Y), r(X,Z)` can be executed with a local selection rule without incurring a suspension if it is called with `Y` ground. In this circumstance, the sub-goals can be executed in the order `p(X,Z)`, `r(X,Z)` and then `q(X,Y)`.

```
:- block p(-, ?).
p(Y, Z) :- Z = true.

:- block q(-, ?), q(?, -).
q(X, Y) :- true.

:- block r(?, -).
r(X, Z) :- X = true.
```

The value of the proposition is that it can be repeatedly applied to show that a compound goal can be executed without suspension under local selection. To see this, observe that following d_i and g_i describe non-suspension requirements and the success patterns for the sub-goals `p(X,Z)`, `q(X,Y)`, `r(X,Z)`. Then:

$$\begin{array}{rlrlrl} d_1 = & \mathtt{Y} & d_2 = & \mathtt{X} \wedge \mathtt{Y} & d_3 = & \mathtt{Z} \\ g_1 = & \mathtt{Z} & g_2 = & \mathrm{true} & g_3 = & \mathtt{X} \\ d_1 \rightarrow g_1 = & \mathtt{Y} \rightarrow \mathtt{Z} & d_2 \rightarrow g_2 = & \mathrm{true} & d_3 \rightarrow g_3 = & \mathtt{Z} \rightarrow \mathtt{X} \end{array}$$

Put $f = \mathtt{Y}$ and observe that $f \wedge \wedge_{i=1}^{3}(d_i \rightarrow g_i) \models \mathtt{X} \wedge \mathtt{Y} \wedge \mathtt{Z} \models (\wedge_{i=1}^{3} d_i)$, or equivalently, $f \models \wedge_{i=1}^{3}(d_i \rightarrow g_i) \rightarrow (\wedge_{i=1}^{3} d_i)$. The proposition thus applies (in the contrapositive direction) hence there must exist $i \in [1, 3]$ such that $f \models d_i$. Indeed $f = \mathtt{Y} \models d_1$, hence the first sub-goal can be executed without suspension.

Now consider $f \wedge g_1$. Since $g_1 \models d_1 \rightarrow g_1$ it follows $f \wedge g_1 \models \wedge_{i=2}^{3}(d_i \rightarrow g_i) \models \wedge_{i=1}^{3} d_i \models \wedge_{i=2}^{3} d_i$. Reapplying the proposition, there must exist $i \in [2, 3]$ such that $f \wedge g_1 \models d_i$. Indeed $f \wedge g_1 \models \mathtt{Y} \wedge \mathtt{Z} \models d_3$, hence the third sub-goal can be executed without suspension.

Now consider $f \wedge g_1 \wedge g_3$. Similarly it follows $f \wedge g_1 \wedge g_3 \models (d_2 \rightarrow g_2) \rightarrow d_2$. Hence by the proposition $f \wedge g_1 \wedge g_3 \models d_2$. Indeed, $f \wedge g_1 \wedge g_3 = \mathtt{X} \wedge \mathtt{Y} \wedge \mathtt{Z} \models d_2$, thus the second sub-goal can next be executed without suspension.

5.4 Monotonic closure of a positive Boolean function

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 the following definition.

Definition 5.8 The maps $\downarrow : Pos_X \to Mon_X$ and $\uparrow : Pos_X \to Mon_X$ are defined by $\downarrow f = \vee\{g \in Mon_X \mid g \models f\}$ and $\uparrow f = \wedge\{g \in Mon_X \mid f \models g\}$.

Example 5.9 *Consider the dyadic case $X = \{x, y\}$ and observe:*

$$\begin{array}{lll} \downarrow(x) = x & \downarrow(x \leftarrow y) = x & \downarrow(x \leftrightarrow y) = x \wedge y \\ \uparrow(x) = x & \uparrow(x \leftarrow y) = \mathrm{true} & \uparrow(x \leftrightarrow y) = \mathrm{true} \end{array}$$

The operation $\downarrow$ arises during analysis and to construct a method for computing $\downarrow$, let $\rho : X \to X'$ be a bijective map where $X' \subseteq Var$ and $X \cap X' = \emptyset$.

Proposition 5.10 *Let $f \in Pos_X$. Then $\downarrow f = \forall_{X'}(d \to \rho(f))$ where $d = \wedge\{x \to \rho(x) \mid x \in X\}$.*

Example 5.11 Consider computing $\downarrow f$ where $X = \{x, y\}$ for the functions $f = x$ and $f = (x \to y)$ of example 5.9. Suppose $\rho(x) = x'$ and $\rho(y) = y'$ so that $d = (x \to x') \wedge (y \to y')$. Then $\downarrow x = \forall_{y'}\forall_{x'} f_1$ where

$$f_1 = d \to x' = ((x \to x') \wedge (y \to y')) \to x'$$

and $\downarrow x = \forall_{y'} f_2$ where

$$\begin{array}{ll} f_2 = f_1[x' \mapsto \mathrm{true}] \wedge f_1[x' \mapsto \mathrm{false}] & \text{hence } \downarrow x = f_2[y' \mapsto \mathrm{true}] \wedge f_2[y' \mapsto \mathrm{false}] \\ \quad = \mathrm{true} \wedge f_1[x' \mapsto \mathrm{false}] & \quad = (x \vee (\neg\mathrm{true})) \wedge (x \vee \neg\neg y) \\ \quad = \neg(\neg x \wedge (y \to y')) = x \vee \neg(y \to y') & \quad = x \wedge (x \vee y) = x \end{array}$$

Now consider $\downarrow(x \to y) = \forall_{x'}\forall_{y'} f_3$ where

$$f_3 = d \to (x' \to y') = ((x \to x') \wedge (y \to y')) \to (x' \to y')$$

and $\downarrow(x \to y) = \forall_{x'} f_4$ where

$$\begin{array}{ll} f_4 = f_3[y' \mapsto \mathrm{true}] \wedge f_3[y' \mapsto \mathrm{false}] & \text{hence } \downarrow(x \to y) = f_4[x' \mapsto \mathrm{true}] \wedge f_4[x' \mapsto \mathrm{false}] \\ \quad = \mathrm{true} \wedge f_3[x' \mapsto \mathrm{false}] & \quad = f_4[x' \mapsto \mathrm{true}] \wedge \mathrm{true} \\ \quad = ((x \to x') \wedge \neg y) \to \neg x' & \quad = \neg(\neg y) = y \end{array}$$

Note that an earlier version of the analysis [22] employed an operator $\circlearrowleft$: $Pos_X \to Pos_X$ that was defined $\circlearrowleft f = \forall_{X'} d \to \rho(f)$ and applied repeatedly. The operator was shown to compute a downward chain which converged onto $\downarrow f$. Although this was not incorrect, profiling experiments revealed that convergence occurred exactly after one application of $\circlearrowleft$. Proposition 5.10 shows that this is no coincidence.

6 Suspension inference analysis

This section formalises the analysis within the framework of abstract interpretation. First, the focus is on the relationship between interpretations and their abstract counterparts. Second, lfp and gfp operators are specified that operate over descriptions of interpretations rather than interpretations themselves. Third, correctness results for these operators are stated (proofs again appear in the appendix).

6.1 Abstract domains

In order to specify how interpretations are abstracted, it is necessary to clarify how sets of substitutions are described as Boolean functions. This is performed with maps α_X and γ_X that specify the relationship between the domain of sets of substitutions and the domain of positive Boolean functions. The maps α_X and γ_X are in turn defined in terms of another α that abstracts a single substitution:

Definition 6.1 Abstraction map for a single substitution $\alpha : Sub \to Def_{Var}$ is defined $\alpha(\theta) = \wedge\{x \leftrightarrow \mathrm{var}(t) \mid x \mapsto t \in \theta\}$.

This map provides a way of abstracting an equation which, in turn, formalises the program abstraction process in which syntactic equations (unifications) are replaced with Boolean functions that capture their grounding behaviour.

Definition 6.2 The abstraction map for a single equation $\alpha : Eqn \to Def_{Var}$ is defined $\alpha(e) = \alpha(\theta)$ where $\theta \in imgu(\{e\})$.

Example 6.3 Let $e = \{g(x, z) = g(f(y, a), y)\}$. Then $\theta_1, \theta_2 \in imgu(\{e\})$ where

$$\theta_1 = \{x \mapsto f(y, a), z \mapsto y\} \qquad \theta_2 = \{y \mapsto z, x \mapsto f(z, a)\}$$

Then $\alpha(\theta_1) = (x \leftrightarrow y) \wedge (y \leftrightarrow z) = \alpha(\theta_2)$, hence $\alpha(e) = (x \leftrightarrow y) \wedge (y \leftrightarrow z)$.

More generally, $\alpha(e)$ is always well-defined for every equation $e \in Equ$ because $\alpha(\theta_1) = \alpha(\theta_2)$ for all $\theta_1, \theta_2 \in imgu(\{e\})$.

Definition 6.4 The abstraction and concretisation maps $\alpha_X : \wp(Sub) \to Pos_X$ and $\gamma_X : Pos_X \to \wp(Sub)$ are defined:

$$\alpha_X(\Theta) = \wedge\{f \in Pos_X \mid \forall\theta \in \Theta.\alpha(\theta) \models f\} \qquad \gamma_X(f) = \{\theta \in Sub \mid \alpha(\theta) \models f\}$$

Example 6.5 Let $X = \{\mathtt{X}, \mathtt{Y}\}$, $\theta_1 = \{\mathtt{X} \mapsto a\}$, $\theta_2 = \{\mathtt{X} \mapsto a, \mathtt{Y} \mapsto b\}$ and $\theta_3 = \{\mathtt{X} \mapsto f(\mathtt{Y}, \mathtt{Z})\}$. Then $\alpha(\theta_1) = \mathtt{X}$, $\alpha(\theta_2) = \mathtt{X} \wedge \mathtt{Y}$ and $\alpha(\theta_2) = \mathtt{X} \leftrightarrow (\mathtt{Y} \wedge \mathtt{Z})$, hence

$$\begin{array}{ccc} \alpha_X(\{\theta_1\}) = \mathtt{X} & \alpha_X(\{\theta_2\}) = \mathtt{X} \wedge \mathtt{Y} & \alpha_X(\{\theta_3\}) = \mathtt{X} \rightarrow \mathtt{Y} \\ \alpha_X(\{\theta_1, \theta_2\}) = \mathtt{X} & \alpha_X(\{\theta_1, \theta_3\}) = \text{true} & \alpha_X(\{\theta_2, \theta_3\}) = \mathtt{X} \rightarrow \mathtt{Y} \end{array}$$

Note that $\alpha_X(\{\theta_1, \theta_3\})$ graphically illustrates the approximate nature of abstraction. The smallest (in fact the only) Boolean function within $Pos_{\{\mathtt{X},\mathtt{Y}\}}$ that describes $\{\theta_1, \theta_3\}$ is true yet this function also represents Sub.

The abstraction map characterises a set of substitutions as a Boolean function that represents all the substitutions in the set and possibly others. In general, the function thus describes a superset of the original set. To conservatively model the predicate $select(p(\vec{x}), \theta)$, however, it is necessary to find a Boolean function that approximates in the other direction, that is, describes a subset of those substitutions under which the predicate holds. The function $select'$ performs just this task:

Definition 6.6 *The function $select' : Atom \rightarrow Mon_{Var}$ is defined by:*

$$select'(p(\vec{x})) = \vee\{f \in Mon_{\vec{x}} \mid \forall \theta \in \gamma_{\vec{x}}(f).\ select(p(\vec{x}), \theta) \textit{ holds}\}$$

Note that $select'$ returns the most general monotonic Boolean function which only describes substitutions that satisfy $select(p(\vec{x}), \theta)$. Relaxing this definition to return a positive Boolean function is problematic: it does not necessarily follow that $select(p(\vec{x}), \theta)$ holds for all $\theta \in \gamma_{\vec{x}}(f_1 \vee f_2)$ if $select(p(\vec{x}), \theta_1)$ holds for all $\theta_1 \in \gamma_{\vec{x}}(f_1)$ and $select(p(\vec{x}), \theta_2)$ holds for all $\theta_2 \in \gamma_{\vec{x}}(f_2)$. To see this, let $f_1 = \mathtt{X}$ and $f_2 = \mathtt{X} \rightarrow \mathtt{Y}$. Then $f_1 \vee f_2 = \text{true}$ and the empty substitution $\epsilon \in Sub = \gamma_{\vec{x}}(f_1 \vee f_2)$ yet $\epsilon \notin \gamma_{\vec{x}}(f_i)$. The following proposition states that this problem is finessed by using monotonic functions, hence $select'$ is well-defined.

Proposition 6.7 The function $select' : Atom \rightarrow Mon_{Var}$ is well-defined.

Example 6.8 Consider again the declaration `:- block append(-, ?, -)` and let $\vec{x} = \langle x_1, x_2, x_3 \rangle$. Recall that $select(\mathtt{append}(\vec{x}), \theta)$ holds iff $\theta(x_1) \notin Var$ or $\theta(x_3) \notin Var$. If $\theta \in \gamma_{\vec{x}}(x_1)$ then $\theta(x_1) \notin Var$ and likewise for the Boolean function x_3. Hence $select(\mathtt{append}(\vec{x}), \theta)$ holds for all $\theta \in \gamma_{\vec{x}}(x_1)$ and for all $\theta \in \gamma_{\vec{x}}(x_3)$. Since $x_1, x_2 \in Mon_{\vec{x}}$ it follows that $select'(\mathtt{append}(\vec{x})) = x_1 \vee x_3$.

6.2 Abstract interpretations

Recall that the fixpoint semantics operates over a computational domain of interpretations. Each interpretation is a set of pairs formed by pairing an atom with a substitution. To realise an abstract version of the fixpoint semantics it is necessary to construct an abstract counterpart of an interpretation. The construction starts with the set of pairs $Base' = \{\langle p(\vec{x}), f\rangle \mid p(\vec{x}) \in Atom \wedge f \in Pos_{\vec{x}}\}$. To order these pairs, let $\vec{x} \leftrightarrow \vec{y} = \wedge_{i=1}^{n}(x_i \leftrightarrow y_i)$ where $\vec{x} = \langle x_1, \ldots, x_n\rangle$ and $\vec{y} = \langle y_1, \ldots, y_n\rangle$. The entailment order on Pos can be extended to $a_1, a_2 \in Base'$ where $a_i = \langle p(\vec{x}_i), f_i\rangle$, $\text{var}(\vec{x}) \cap \text{var}(\vec{x}_i) = \emptyset$ and $f'_i = \exists_{\vec{x}_i}((\vec{x} \leftrightarrow \vec{x}_i) \wedge f_i)$ by defining $a_1 \models a_2$ iff $f'_1 \models f'_2$. Note that the ordering cannot merely be defined by $a_1 \models a_2$ iff $f_1 \models f_2$ since f_1 and f_1 can range over different sets of variables. The abstract domain of interpretations is then given by $Int' = \{I' \subseteq Base' \mid down(I') = I'\}$ where $down(I') = \{a_1 \in Base' \mid \exists a_2 \in I'.a_1 \models a_2\}$. As in the concrete setting, the $\langle Int', \subseteq, \cup, \cap, Base', \emptyset\rangle$ is a complete lattice.

The following definition extends α and γ to interpretations and thereby completes the domain construction by relating interpretations with their abstractions.

Definition 6.9 The abstraction and concretisation maps $\alpha : Int \to Int'$ and $\gamma : Int' \to Int$ are defined:

$$\gamma(I') = \{\langle p(\vec{x}), \theta\rangle \mid \langle p(\vec{x}), f\rangle \in I' \wedge \alpha(\theta) \models f\} \quad \alpha(I) = \cap\{I' \in Int' \mid I \subseteq \gamma(I')\}$$

6.3 Specification of the lfp

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

Definition 6.10 Given a logic program P, the abstract fixpoint operator $\mathcal{F}' : Int' \to Int'$ is defined by:

$$\mathcal{F}'(I') = down\left\{ \langle p(\vec{x}), g\rangle \;\middle|\; \begin{array}{c} p(\vec{x}) \text{ :- } e, p_1(\vec{x}_1), \ldots, p_n(\vec{x}_n) \in P \wedge \\ \langle p_i(\vec{x}_i), g_i\rangle \in I' \qquad\qquad \wedge \\ g = \overline{\exists}_{\vec{x}}(\alpha(e) \wedge \wedge_{i=1}^{n} g_i) \end{array} \right\}$$

Application of the $down$ function ensures that the operator $\mathcal{F}'$ yields a result that appears within Int'. Without it the operator could produce a set I' that included a pair $\langle p(\vec{x}), g\rangle$ but excluded $\langle p(\vec{y}), \exists_{\vec{x}}((\vec{x} \leftrightarrow \vec{y}) \wedge g)\rangle$, hence $I' \neq down(I')$.

The operator $\mathcal{F}'$ is continuous, hence an abstract fixpoint semantics can be defined $\mathcal{F}'(P) = \text{lfp}(\mathcal{F}')$. Moreover, $\text{lfp}(\mathcal{F}')$ can be computed in an iterative manner by calculating the limit of the increasing sequence $\emptyset$, $\mathcal{F}'(\emptyset)$, $\mathcal{F}'(\mathcal{F}'(\emptyset))$, etc. The limit can be finitely calculated because Int' contains a finite number of elements since (1) the number of predicate symbols in any program P is finite and (2) Pos_X is finite for any finite variable set X. The following correctness asserts that $\mathcal{F}'$ faithfully characterises $\mathcal{F}$ with respect to γ.

Theorem 6.11 $\mathcal{F}(P) \subseteq \gamma(\mathcal{F}'(P))$.

6.4 Specification of the gfp

The central component of the suspension analysis is the gfp calculation that infers calling patterns for predicates which are sufficient for non-suspension under local selection. Recall that the syntactic structure of call patterns coincide with that of success patterns. Hence the domain of interpretation descriptions Int', that provides the basis for $\mathcal{F}'$, also provides a basis for the operator $\mathcal{B}'$ that prescribes the gfp calculation. $\mathcal{B}'$ is defined for a clause, then a predicate, then a program.

Definition 6.12 The operator $\mathcal{B}' : Int' \to Int'$ is defined:

$$\mathcal{B}'_{p(\vec{x}):\text{-}b}(I') = down \left\{ \langle p(\vec{x}), d' \rangle \;\middle|\; \begin{array}{r} b = e, p_1(\vec{x}_1), \ldots, p_n(\vec{x}_n) \;\wedge \\ \langle p_i(\vec{x}_i), g_i \rangle \in \mathrm{lfp}(\mathcal{F}') \qquad \wedge \langle p_i(\vec{x}_i), d_i \rangle \in I' \wedge \\ g = \alpha(e) \wedge (\wedge_{i=1}^{n} d_i \to g_i) \wedge \qquad d = \wedge_{i=1}^{n} d_i \wedge \\ d' = select'(p(\vec{x})) \wedge \;\downarrow (\overline{\forall}_{\vec{x}}(g \to d)) \end{array} \right\}$$

$$\mathcal{B}'_{p(\vec{y})}(I') = \cap\{\mathcal{B}'_{p(\vec{x}):\text{-}b}(I') \mid \|\vec{x}\| = \|\vec{y}\| \wedge p(\vec{x}) \;\text{:-}\; b \in P'\}$$

$$\mathcal{B}'(I') = \cup\{\mathcal{B}'_{p(\vec{y})}(I') \mid p(\vec{y}) \in Atom\}$$

Consider the operator $\mathcal{B}'_{p(\vec{x}):\text{-}b}$ for a clause $p(\vec{x})\text{:-}b$ with body b. Recall that the Boolean function $g \to d$ captures a grounding condition that, if satisfied by the state, is sufficient for executing b without suspension under local selection. Put another way, if $\alpha(\theta) \models g \to d$ then $\langle b, \theta \rangle$ will not suspend under local selection. The function $d' = select'(p(\vec{x})) \wedge {\downarrow} (\overline{\forall}_{\vec{x}}(g \to d))$, entails $g \to d$, hence is also sufficient for non-suspension. The function d' additionally satisfies the properties that: (1) it is only couched in terms of the variables $\vec{x}$, (2) it is monotonic and (3) it guarantees that $select'(p(\vec{x}), \theta)$ holds. The operator $\mathcal{B}'_{p(\vec{y})}$ conjoins the formulae calculated from the individual clauses of the predicate to obtain a formula sufficient for non-suspension no matter what clause is selected (the constraint $\|\vec{x}\| = \|\vec{y}\|$ ensures that the arity of the clauses match those of the predicate). The operator $\mathcal{B}'$ calculates non-suspension conditions for each predicate in the program.

The value of the operator $\mathcal{B}'$ is explained by the following theorem. The theorem states that $Base', \mathcal{B}'(Base'), \mathcal{B}'(\mathcal{B}'(Base'))$, etc respectively characterise sets of initial states for which any suspension occurring under local selection can only manifest itself after at least 0, 1, 2 etc steps.

Theorem 6.13 Let $\langle p(\vec{z}), \theta\rangle \in \gamma({\mathcal{B}'}^k(Base'))$. If $\langle p(\vec{z}), \theta\rangle {\twoheadrightarrow_{local}}^l susp$ then $l \geq k$.

The operator $\mathcal{B}'$ is co-continuous, hence $\text{gfp}(\mathcal{B}')$ exists. Moreover, since Int' is finite, $\text{gfp}(\mathcal{B}')$ can be calculated as the limit of the sequence, $Base'$, $\mathcal{B}'(Base')$, $\mathcal{B}'(\mathcal{B}'(Base'))$, etc. The significant of the limit is that it describes a set of states that can never generate a suspend under local selection. The immediate corollary is that the same states can never lead to a suspension when the local selection requirement is dropped. This is the correctness result that underpins that analysis.

Corollary 6.14 If $\langle p(\vec{z}), \theta\rangle {\twoheadrightarrow_{delay}}^k susp$ then $\langle p(\vec{z}), \theta\rangle \notin \gamma(\mathcal{B}'(P))$.

7 Implementation

One advantage that the suspension analysis has over many of its predecessors is the simplicity with which it can be implemented. This section explains how the analysis can be realised as two meta-interpreters: one for computing an approximation of the success set (which computes the lfp); and the other for performing the backwards analysis itself (which computes the gfp). The meta-interpreters are presented as Prolog programs and, for generality, these programs are parameterised by predicates that realise operations on Boolean functions. In actuality, Boolean functions might be represented as BDDs [1] or as lists of clauses [29] or some other way. These predicates are interpreted as follows:

- `bool_and(`φ_{in_1}`, `φ_{in_2}`, `φ_{out}`)` succeeds whenever φ_{in_1} and φ_{in_2} are bound to data-structures representing Boolean functions and subsequently binds φ_{out} to a data-structure that represents $\varphi_{in_1} \wedge \varphi_{in_2}$;
- `bool_or(`φ_{in_1}`, `φ_{in_2}`, `φ_{out}`)` binds φ_{out} to $\varphi_{in_1} \vee \varphi_{in_2}$;
- `bool_implies(`φ_{in_1}`, `φ_{in_2}`, `φ_{out}`)` binds φ_{out} to $\varphi_{in_1} \rightarrow \varphi_{in_2}$;

- `bool_entails(`φ_{in_1}`, `φ_{in_2}`)` is a test predicate that neither instantiates φ_{in_1} nor φ_{in_2}; it succeeds if and only if $\varphi_{in_1} \models \varphi_{in_2}$;
- `bool_iff(X, Xs, `φ_{out}`)` succeeds whenever `X` is a variable and `Xs` is a list of variables `[`$\texttt{X}_1$`, ..., `$\texttt{X}_n$`]`, whereupon the predicate instantiates φ_{out} to a data-structure that represents the Boolean function $\texttt{X} \leftrightarrow (\wedge_{i=1}^{n} \texttt{X}_i)$;
- `bool_rename(`φ_{in}`, Xs, Ys, `φ_{out}`)` succeeds whenever φ_{in} represents a Boolean function and `Xs` and `Ys` are lists of variables $[\texttt{X}_1, \ldots, \texttt{X}_n]$ and $[\texttt{Y}_1, \ldots, \texttt{Y}_n]$, whence each occurrence of $\texttt{X}_i$ in φ_{in} is replaced with $\texttt{Y}_i$ to obtain φ_{out};
- `bool_monotonic(`φ_{in}`, `φ_{out}`)` binds φ_{out} to the most general Boolean function which is both monotonic φ_{out} and $\varphi_{out} \models \varphi_{in}$;
- `bool_universal(`φ_{in}`, `t`, `φ_{out}`)` applies universal quantifier elimination to project φ_{in} onto those variables occurring in the term t to obtain φ_{out};
- `bool_existential(`φ_{in}`, `t`, `φ_{out}`)` applies existential quantifier elimination to project φ_{in} onto the variables in t to obtain φ_{out}.

The meta-interpreters operate on programs stored as database of facts of the form `my_clause`$(h(\vec{x}), [b_1, \ldots, b_n])$. Each body atom b_i is either a call $p_i(\vec{x}_i)$ to a predicate that is defined within the program or an `iff(`x`, [`$x_1, \ldots, x_m$`])` atom. The $p_i(\vec{x}_i)$ atoms are flat, that is, their arguments $\vec{x}_i$ are distinct variables. Grounding dependencies between arguments, and more generally the variables occurring in a clause, are recorded by `iff(`x`, [`$x_1, \ldots, x_m$`])` atoms that represent $x \leftrightarrow (\wedge_{i=1}^{n} x_i)$ dependencies [7]. For example, the grounding behaviour of the unification $x = t$ is modelled by `iff(`x`, [`$x_1, \ldots, x_m$`])` where $var(t) = \{x_1, \ldots, x_m\}$. More generally, a unification $t_1 = t_2$ is modelled by a composition `iff(`x`, [`$x_1, \ldots, x_m$`])`, `iff(`x`, [`$y_1, \ldots, y_n$`])` where $var(t_1) = \{x_1, \ldots, x_m\}$, $var(t_2) = \{y_1, \ldots, y_n\}$ and x is a fresh variable. Strategies for modelling other Prolog builtins are documented in [27]. This approach, which essentially compiles builtins into combinations of `iff` atoms, encapsulates the complexity of handling builtins into a single module that precedes the analysis itself. It also simplifies, and thereby speeds up, the meta-interpreters. In addition to the `my_clause` database, an `assertion(`$p(\vec{x})$`, `φ`)` database records a Boolean function for each predicate that is sufficient for a call $p(\vec{x})$ to not suspend (immediately). For example, for the block declaration `:- block p(-, ?,`

-), p(?, -, -) specifies that the predicate p will suspend until either its first and second arguments or its third argument is instantiated. The table would therefore include a fact assertion($p(\vec{x})$, φ) in which φ represents the Boolean function $(x_1 \wedge x_2) \vee x_3$.

Figure 2 lists the two interpreters. Both interpreters manipulate a memo table of facts of the form memo(lfp, $p(\vec{x}), \varphi$) and memo(gfp, $p(\vec{x}), \varphi$) where φ represents a Boolean function. These facts are added and removed from the memo table to record the status of the lfp and the gfp. For illustrative purposes, both meta-interpreters implement a simple (Gauss-Seidel [12]) iteration strategy, that is, reapply all clauses until stability is achieved. A call to lfp_iterate triggers another iteration of lfp_operator. Whenever the memo table is updated, a flag is raised in lfp_mutate. Iteration ceases when retract(flag) fails in the second clause of lfp_iterate, that is, when no changes where made in the previous iteration. The following two sections describe the predicates that implement the lfp and the gfp calculations.

7.1 Implementation of the lfp

In the case of the lfp, the memo(lfp, $p(\vec{x})$, φ) facts stores a series of Boolean functions $\varphi_1, \varphi_2, \ldots$, that for a given $p(\vec{x})$, constitute an increasing sequence (chain) over the lifetime of the lfp calculation. If the memo table does not contain a fact for $p(\vec{x})$, then φ is considered to be false. This saves initialising the table with facts of the form memo(lfp, $p(\vec{x})$, false).

The lfp_operator predicate looks up a clause in the my_clause(Head, Body) database and proceeds to solve the Body of that clause using the Boolean functions currently stored in the memo table. This is realised by lfp_solve(Body,φ_{in},φ_{out}) which conjoins φ_{in} with a Boolean function for each atom in Body to obtain φ_{out}. If Body is empty then $\varphi_{in} = \varphi_{out}$. Otherwise the first atom in Body is either an iff(X, [X$_1$, ..., X$_n$]) atom or an atom $p(\vec{x})$ defined elsewhere in the program. In the former case, the accumulated Boolean function that is used to solve the remaining body atoms is $\varphi_{in} \wedge (\mathtt{X} \leftrightarrow (\wedge_{i=1}^{n} \mathtt{X}_i))$. In the latter case, it is $\varphi_{in} \wedge \varphi_1$ where φ_1 is the current value for $p(\vec{x})$ in the memo table. Note that if the table omits a memo(lfp, $p(\vec{x})$, φ) fact, then get_memo(lfp, Atom, φ_1) fails, hence lfp_solve fails and the memo table is unchanged. The remaining calls of lfp_operator project the resulting Boolean function onto the variables of Head and, if necessary, modify the memo table with the projection.

The predicate lfp_update(Head, φ_{in}), if necessary, updates the function stored in the memo table for Head. If the table contains a fact memo(lfp, Head, φ) and $\varphi_{in} \models \varphi$, then no update is necessary. Otherwise,

```
lfp_iterate :- lfp_operator, fail.
lfp_iterate :- retract(flag), lfp_iterate.
lfp_iterate.

lfp_operator :-
   my_clause(Head, Body),
   lfp_solve(Body, true, φ1),
   bool_existential(φ1, Head, φ2),
   lfp_update(Head, φ2).

lfp_solve([], φ, φ).
lfp_solve([iff(X,Xs)|Atoms],φin,φout) :-
   bool_iff(X, Xs, φ1),
   bool_and(φin, φ1, φ2),
   lfp_solve(Atoms, φ2, φout).
lfp_solve([Atom|Atoms], φin, φout) :-
   get_memo(lfp, Atom, φ1),
   bool_and(φin, φ1, φ2),
   lfp_solve(Atoms, φ2, φout).

lfp_update(Head, φin) :-
   (memo(lfp, Head, φ1),
    bool_entails(φin, φ1) ->
       true
   ;
       lfp_mutate(Head, φin)
   ).

lfp_mutate(Head, φin) :-
   (retract(memo(lfp, Head, φ1)) ->
       bool_or(φin, φ1, φ2)
   ;
       φ2 = φin
   ),
   assert(memo(lfp, Head, φ2)),
   (flag ->
       true
   ;
       assert(flag)
   ).

get_memo(Type, Atom, φout) :-
   functor(Atom, Name, Arity),
   functor(Fresh_Atom, Name, Arity),
   Atom =..[Name, Args],
   Fresh_Atom =..[Name, Fresh_Args],
   memo(Type, Fresh_Atom, φ),
   bool_rename(φ, Fresh_Args, Args, φout).
```

```
gfp_iterate :- gfp_operator, fail.
gfp_iterate :- retract(flag), gfp_iterate.
gfp_iterate.

gfp_operator :-
   my_clause(Head, Body),
   gfp_solve(Body, true, true, φ1),
   assertion(Head, φ2)
   bool_and(φ1, φ2, φ3),
   bool_universal(φ3, Head, φ4),
   bool_monotonic(φ4, φ5),
   gfp_update(Head, φ5).

gfp_solve([], ϕin, ψin, φout) :-
   bool_implies(ϕin, ψin, φout).
gfp_solve([iff(X,Xs)|Atoms],ϕin, ψin, φout) :-
   bool_iff(X, Xs, φ1),
   bool_and(ϕin, φ1, ϕ1),
   gfp_solve(Atoms, ϕ1, ψin, φout).
gfp_solve([Atom|Atoms], ϕin, ψin, φout) :-
   get_memo(gfp, Atom, φ1),
   get_memo(lfp, Atom, φ2),
   bool_implies(φ1, φ2, φ3),
   bool_and(ϕin, φ3, ϕ1),
   bool_and(ψin, φ1, ψ1),
   gfp_solve(Atoms, ϕ1, ψ1, φout).

gfp_update(Head, φin) :-
   (memo(gfp, Head, φ1),
    bool_entails(φ1, φin) ->
       true
   ;
       gfp_mutate(Head, φin)
   ).

gfp_mutate(Head, φin) :-
   (retract(memo(gfp, Head, φ1)) ->
       bool_and(φin, φ1, φ2)
   ;
       φ2 = φin
   ),
   assert(memo(gfp, Head, φ2)),
   (flag ->
       true
   ;
       assert(flag)
   ).
```

Figure 2: Meta-interpreters for computing the lfp (left) and the gfp (right)

`lfp_mutate(Head,` φ_{in}`)` is invoked which replaces `memo(lfp, Head,` φ`)` (if it exists) with the new fact `memo(lfp, Head,` $\varphi_{in} \vee \varphi$`)`. The predicate also ensures that the `flag` is raised.

One technicality that complicates analysis is renaming. The problem is that variables occurring in a body atom do not necessary concur with those in the memo table. For example, if BDDs were used to represent Boolean functions then, most likely, each `memo(lfp,` $p(\vec{x})$`,` φ`)` fact would store an atom $p(\vec{x})$ in which $\vec{x}$ was a sequence of distinct numbers. Then φ would point to a BDD over propositional variables identified with these numbers. Under this scheme, variables occurring in each clause would also be numeric. The numbers occurring in the arguments of a body atom would not in general match those in the table for that predicate, hence the requirement for renaming. Renaming is thus performed as formulae are retrieved from the table by `get_memo(Type, Atom,` φ_{out}`)`. This predicate is used in both the lfp and gfp calculation; `Type` indicates whether a `lfp` or `gfp` fact is to be read.

7.2 Implementation of the gfp

Much of the gfp interpreter mirrors that of the lfp interpreter. The iteration strategy is analogous, except that the `memo(gfp,` $p(\vec{x})$`,` φ`)` facts define a decreasing chain $\varphi_1, \varphi_2, \ldots$ over the history of the gfp calculation. Moreover, if the table does not contain a fact for $p(\vec{x})$, then φ is considered to be true. Again, this saves on initialisation.

The `gfp_operator` predicate considers each clause in `my_clause(Head, Body)` individually and solves the body atoms in `Body` using Boolean functions extracted from the pre-computed lfp memo table and the partially computed gfp memo table. The predicate `gfp_solve(Body,` ϕ_{in}`,` ψ_{in}`,` φ_{out}`)` iterates over the list `Body` accumulating new Boolean functions, ϕ'_{in} and ψ'_{in} say, as each atom in `Body` is considered. Iteration ceases when the body is empty, whereupon φ_{out} is instantiated to the function $\phi'_{in} \to \psi'_{in}$. When `Body` has been traversed, ψ'_{in} is the conjunction of those φ_1 in the gfp memo table that match an atom in `Body`. Likewise, ϕ'_{in} is also a conjunction. It is the conjunction of formulae of the form $\varphi_1 \to \varphi_2$ where φ_1 and φ_2 respectively match body atoms in the gfp and lfp tables.

Once the function $\phi'_{in} \to \psi'_{in}$ is computed for `Body`, the result is conjoined with the assertion, and universal variable elimination is applied to remove any variables not occurring in `Head`. Finally, the call to `bool_monotonic` strengthens (if necessary) the projection to ensure that it is monotonic.

It is straightforward to refine these meta-interpreters to incorporate semi-naive iteration [54], that is, only reapply those clauses which contain an atom whose memo value was modified in the previous iteration. Another refinement to compute the strongly connected components (SCCs) of the call graph of a given program, topologically order the SCCs, and then iterate and stablise on lower SCCs before proceeding to higher SCCs. An experimental analyser that incorporates these two refinements can be found at:

`http://www.cs.kent.ac.uk/people/staff/amk/susweb.html`

The analyser uses the PiLLoW library [25] for an HTML interface. The above URL provides all the benchmarks used within the experiments, enables the analysis to be rerun on these examples, whilst providing an interface for experimenting with the analysis on new examples.

8 Experimental evaluation

To assess the analysis, it was applied to a series of programs solicited from various sources. Figure 3 summarises the precision and timing results where

- the *preds* column indicates the number of predicates occurring in the programs (ignoring any builtin predicates);
- *blocks* gives the number of predicates for which call patterns other than false were inferred;
- *%* denotes the ratio of *blocks* to *preds* expressed as a percentage;
- *abs* is the time required to read the file off the disk, parse it and preprocess it into pure Horn clauses ready for analysis;
- *SCC* records the time necessary for computing the strongly connected components (SCCs) of the call graph of the program [48];
- the columns *lfp* and *gfp* record the times required to compute the least and greatest fixpoints respectively;
- *total* records the total analysis time that additionally includes any overhead for pretty printing the results to HTML files [25].

The timings experiments were conducted on a PC with a Intel Pentium III 933MHz processor, 640MB memory, running SICStus 3.7.1 on Linux 2.6.7.

source	program	*precision* preds	blocks	%	*time (msecs)* abs	SCC	lfp	gfp	total
[15]	dnf	8	8	100	11	1	2	3	17
	combo	10	10	100	9	2	2	3	16
	transp	11	11	100	12	2	1	4	19
	primes	7	7	100	12	1	1	2	16
	bessel	16	16	100	12	2	1	5	20
	deriv	7	7	100	9	1	1	2	13
[19]	sieve	6	6	100	7	1	1	5	14
	insert	8	8	100	9	1	1	2	13
	btree	10	10	100	11	2	2	4	19
	ssd	24	24	100	33	5	8	18	64
[26]	hamming	6	6	100	8	1	2	3	14
[29]	entails	8	8	100	7	1	1	3	12
[30]	colouring	42	37	88	42	9	30	40	121
	spanning	76	71	93	81	28	84	153	346
	eight_puzzle	97	88	91	100	40	75	211	426
[32]	PTMddd	319	316	99	476	592	785	25138	26991
[38]	queens	10	10	100	7	1	1	3	12
[40]	msort_control	14	14	100	14	3	3	5	25
	qsort_control	11	11	100	11	2	1	5	19
	queens_control	15	15	100	13	3	2	4	22
[50]	bestpath	20	11	55	39	4	11	19	73
	fact	7	7	100	7	1	1	1	10
	isotrees	20	20	100	16	3	3	7	29
	pascal	23	23	100	22	5	5	8	40
	mm	19	19	100	17	3	5	5	30
	hanoi	8	8	100	7	2	2	2	13
	msort	23	23	100	18	4	6	9	37
	semigroup	20	19	95	21	4	6	11	42
	mastermind	20	20	100	28	5	6	148	187
	nand	25	25	100	32	6	8	15	61

Figure 3: Summary of precision and timing results

All timings were averaged over 10 runs. Note that the on-line analyser is implemented on slightly slower machine and its timings can vary with load.

Many of the programs used for experiments were adapted from programs written in concurrent logic languages, such as Strand [19], KL1 [49] or Parlog

[26], or the distributed constraint programming language Janus [15]. All the programs denotated by Foster, Huntbach and Johnson were coded in Strand; all the programs denotated by Debray were coded in Janus. Many of the KL1 programs denotated by Tick and Massey were tuned for efficient parallel evaluation [49]. All the synchronisation in these programs was reexpressed in Prolog with block declarations and the resulting programs tested. The analysis was then applied to these 30 programs.

In terms of efficiency, the total analysis times are sufficiently low for the analysis to be applied repeatedly and frequently during the development of an application. For the larger benchmarks, computing the gfp dominates the cost of the analysis. This is because the cost of abstracting the file and computing the SCCs, grows linearly with the size of the program. Calculating the gfp is more expensive than computing the lfp, partly because of the cost of calculating monotonic closure $\downarrow$, partly because the function $d_i \rightarrow f_i$ can be represented less densely than f_i, and partly because the lfp can be optimised with some BDD refinements that are not applicable to the gfp [46]. It is worth pointing out that the times to compute the lfp and the gfp would benefit significantly from replacing the BDD library of Armstrong and Schachte [1] with a faster and more space efficient version [2]. It is more difficult to quantatively qualify the usefulness of results generated by the analysis and therefore the results are discussed in the following sections. Section 8.1 reports some discrepancies that were discovered by the analysis. However, since the analysis is based on approximation, it will inevitably not be able to infer some queries for which the program will actually not suspend. This leads to the issue of false positives – if too many of the false call patterns arise from imprecision (rather than genuine coding errors), then the usefulness of the analysis is compromised. Section 8.2 thus catalogs the false positives that arose in the analysis of the programs. This section also explains those circumstances, for example which coding styles, are problematic for analysis. Section 8.3 considers applying the analysis to code generated by program transformation, namely, through control generation. This section demonstrates how suspension analysis can throw new light on the termination inference problem [21, 43] – the problem of inferring a class of terminating goals for a given program. Section 8.4 explains the debugging techniques that were found to be useful for applying suspension inference to a larger application.

8.1 Anomalies discovered through the analysis

8.1.1 bessel

For this program, the analysis inferred a call pattern of false for the predicate `bessel`, the problem stemming from the clause:

```
bessel(0, X, Y1, Y2) :- Y2 = 0.0, j0(10, X, Y).
```

According to a comment in the original source, `bessel(N, X, Y1, Y2)` should bind `Y1` and `Y2` to the values of the functions $J_N(X)$ and $J_{N-1}(X)$. However, `Y1` is never instantiated, leading to a suspension whenever this clause is selected. The author of bessel confirmed that the goal `j0(10, X, Y)` should be `j0(10, X, Y1)` and this amendment was sufficient to verify complete non-suspension.

8.1.2 ssd

For this program, a call pattern of false was inferred for the predicate that is given below:

```
try_orient( [], Remain, Pent, [Orient, Starts, Pattern | Orients],
        Board, First, History, D) :-
    member( First, Starts, Mem),
    start_match( Mem, Pattern, Board, New_Board, D1),
    add_pent( D1, Remain, Pent, Orient, New_Board, History, D2),
    try_orient( D2, Remain, Pent, Orients, Board, First, History, D).
try_orient( [], _, _, [], _, _, _, D) :- D := [].
```

The processes `start_match`, `add_pent` and `try_orient` bind the variables `D1`, `D2` and `D` to `[]` indicate termination. The processes `add_pent` and `try_orient` suspend until `D1` and `D2` are bound and thus these "done" variables act to serialise the computation. The suspension arises because `add_pent` is defined in terms of a predicate `next_play`, listed below, whose final clause does not instantiate `D`. The original author of ssd confirmed this was a bug, indeed, the oversight seems to relate to the commented out case. The commented out case prints a message and, in doing so, `print_history` sets `D`. Yet when printing was deactivated, by replacing the commented out clause with the last clause, `D` was left unset.

```
next_play( Remaining, Board, History, D) :- Remaining =\= [] |
    length( Board, Len),
```

```
    First is (2 * Len) // 3,
    try_pent( [], Remaining, Remaining, Board, First, History, D).
%next_play( [], _, History, D) :-
%   print_history( " SOLN ", History, D).
next_play( [], _, History, D).
```

8.1.3 queens_control

An anomaly was spotted within this program, not because a call pattern of false was inferred, but because the analysis only inferred that a certain predicate, `perm`, will not suspend if its first argument is ground. Curiously the control for `perm` was generated with a technique [40] that ensures that the predicate will not suspend if either its first or second argument are ground. The discrepancy was tracked to the following declaration for `perm_aux`:

```
:- block perm_aux(-, ?, ?). perm_aux(?, -, ?).
perm_aux(D1, D2, D) :- D1 = D2, D = D1.
```

The original intention behind the declaration was to suspend until its first *and* second arguments were ground. The separator between `perm_aux(-, ?, ?)` and `perm_aux(?, -, ?)`, however, should be a comma. The period is an unfortunate error since it introduces a clause `perm_aux(?, -, ?)` which, though spurious, is not syntactically incorrect.

What is interesting is that none of these problems manifested themselves during testing, either because of the case coverage within the test harnesses or because of the particular interleavings adopted by the scheduler.

8.2 False positives generated by the analysis

8.2.1 semigroup and eight_puzzle

For the program semigroup, non-suspension could only not be shown for the top-level predicate `main`:

```
main(N) :-
    kernel(K),
    append([begin|K],[end|R],S),
    spawn(S,R,Out,[]),
    count(Out,N).
```

The analysis infers that `spawn(S,R,Out,[])` will not suspend if both `S` and `R` are ground. However, `spawn` implements a form of pipelined filter where the input stream S is fed by the output stream R [49]. Thus neither S nor R are ground at the time of the call (though kernel(K) binds K to a ground structure).

A similar cyclic dependency between streams was discovered in the program eight_puzzle in the following clause:

```
search1(false, Upbound, State, Moves, PCosts, Sols, Count1) :-
    search1_aux(Count, Count1),
    gen_succs(State, Moves, Succs),
    succs_search(Succs, [Upbound | NCosts], CSols, Count),
    manage_children(State, Upbound, PCosts, CSols, NCosts, Sols).
```

The stream CSols that is output from the succs_search process is input into manage_children process and conversely the stream NCosts that is output from succs_search is input into manage_children. Note the streams S and R are initially open-ended (likewise CSols and NCosts), and abstracting these streams as rigid lists (or even dependencies between rigid lists [23]) is not sufficient for analysing this class of problem.

8.2.2 colouring, spanning and bestpath

A call pattern of false for the predicate setup in the program colouring was traced to use of the back-communication protocol [19] in the following predicate:

```
send_colour(Colour,Accepted,[I/O|Channels]) :-
    O:=[mess(Colour,Accepted)|Channel],
    send_colour(Colour,Accepted,Channels).
```

A send_colour process will suspend until its third argument is bound to a list whose first element is an I/O term. The variable O is then instantiated. The analysis infers that send_colour will not suspend if it is called with a ground third argument. However, this sufficient condition is never satisfied, indeed send_colour is responsible for partially instantiating this argument. Similar use of back-communication impeded full analysis of the program spanning (indeed spanning was derived from colouring [30]). Back-communication was also responsible for false positives in bestpath, the problem emanating from the clause:

```
eval([echo(X)|Next],BestCost,Paths,BestPath) :- true |
    X = BestCost,
    eval(Next,BestCost,Paths,BestPath).
```

Note that although this style of programming introduces false positives, it is interesting to see that for these programs, the analysis can pinpoint the lowest SCC which relies on this protocol. This is useful since knowing where this protocol is applied, aids the understanding of code developed by a third-party.

8.3 A case study on applying suspension analysis with control generation

The objective of control generation is to derive a computation rule for a set of Horn clauses which is efficient, assures termination and yet is complete, that is, it computes all answers to a given query [4, 37, 45]. Progress in solving this fundamental problem in logic programming has been slow, but recently it has been shown how control generation can be tackled by program transformation [40]. The transform relies on information about the depths of derivations to derive block declarations which orchestrate the control. The relevance for suspension analysis is twofold:

- the transformation introduces blocks that allow unusually flexible, albeit unusually subtle, control. The resulting code is a particular challenge for analysis. For example, applying the transform to a mergesort [40] gives a program that can be executed both in forward mode to sort a list into ascending order and backward mode to permute an ordered list.
- the transform is guaranteed to generate programs that either suspend or universally terminate, that is, finitely enumerate all the answers to a particular goal [52]. Hence, if suspension analysis can infer a class of goals for which the transformed program does not suspend, then universal termination is assured for this class of query. Hence, through the application of program transformation, suspension inference can be adapted to tackle the termination inference problem [21, 43] – the problem of inferring a class of terminating goals for a given program.

The control generation transform was applied to three programs: mergesort, quicksort and queens (see [40] for further details) and the resulting code analysed. The first predicate of mergesort that was found to have a call pattern of false was `split`:

```
split(L, L1, L2) :-
    split_aux(S, S1, S2, D),
```

```
    list_length(L, S, D),
    list_length(L1, S1, D),
    list_length(L2, S2, D),
    split_sdr(L, L1, L2, D).
```

The predicate `split` is itself defined in terms of `list_length`, `split_aux` and `split_sdr`. The predicate `list_length(L, S, D)` delays until `L` is bound to a rigid list, that is, a list of determined length, before instantiating `S` to the length of `L`. The predicate `split_aux(S, S1, S2, D)` suspends until either `S`, `S1` or `S1` are instantiated. Then `D` is bound to either `S`, `2 * S1` or `2 * S2 + 1`. The predicate `split_sdr(L, L1, L2, D)` suspends until `D` is instantiated. The call patterns generated for `list_length`, `split_aux` and `split_sdr` where as expected. On closer inspection, however, it was found that the success set of `list_length` was merely true. By repeatedly commenting out the clauses of the auxiliary `list_length_aux` and rerunning the analysis, the first clause was diagnosed as the source of the problem. This clause is designed to abort an unnecessary list traversal and does not instantiate any output arguments. For example, if `L` is rigid, then the goal `list_length(L, S, D)` will bind `S`, which is sufficient to instantiate `D`. Hence there is no need to traverse `L1` and `L2`; these calls to `list_length` can be aborted.

```
list_length(Xs, L, Kill) :- list_length_aux(Xs, 0, L, Kill).

:- block list_length_aux(-, ?, ?, -).
list_length_aux(_, _, _, Kill) :-
    nonvar(Kill), !.
list_length_aux([], L, L, _).
list_length_aux([_ | Xs], A, L, Kill) :-
    A1 is A + 1,
    list_length_aux(Xs, A1, L, Kill).
```

This clause is problematic for analysis and for consistency, to ensure that `list_length` always instantiated its second argument, the first clause of `list_length_aux` was replaced with:

```
list_length_aux(_, _, L, Kill) :-
    nonvar(Kill), !,
    L = dummy.
```

With this revision in place, the analysis inferred that `split(L, L1, L2)` will not suspend if it is called with either `L`, `L1` or `L2` are ground. The necessity of this revision suggests that suspension analysis is unlikely to be completely automatic for programs with such sophisticated control. This amendment was sufficient to infer non-trivial call patterns for the rest of mergesort. In particular the analysis verified that mergesort can operate in two modes without incurring suspension. Moreover, by virtue of the correctness of the program transformation [40], it follows that if either the first or second arguments the top-level query are ground, then the query is also guaranteed to terminate. This is an interesting result within itself since most termination inference schemes [21, 43] presuppose left-to-right control.

8.4 A case study on applying suspension analysis to a large application

PTMddd is a tableau theorem prover developed at the Manchester Metropolitan University by Fisher and Johnson that applies a form of parallel depth-first search [32]. The Strand version of the program exceeds 4000 LOC which define 319 predicates over 14 modules. The program is challenging computationally for analysis since it contains a large SCC of 16 mutual recursive predicates each of which has between 11 and 16 arguments. The results for each predicate were then inspected one-by-one starting the predicates at the lowest nodes in the call graph. The first predicate with a call pattern of false was `set_rf4`; the SCC algorithm that precedes analysis places this predicate in SCC number 78 out of the 277 SCCs occurring in the program. The Strand code for this predicate is given below:

```
set_rf4(PiSet,{Dk,Tk},true,true,_,Flag,NewPiSet):-
    Flag := true,
    lhs_strip_DmTm(PiSet,[],Dk,Tm,NewPiSet1),
    compose(NewPiSet1,{Dk,Tk},[],NewPiSet2),
    compose(NewPiSet2,{Tm,Tk},[],NewPiSet).
set_rf4(PiSet,{Dk,Tk},false,true,true,Flag,NewPiSet):-
    Flag := true, rhs_strip_DmTm(PiSet,[],Dk,Dm,NewPiSet1),
    compose(NewPiSet1,{Dk,Tk},[],NewPiSet2),
    compose(NewPiSet2,{Dm,Tk},[],NewPiSet).
set_rf4(PiSet,_,_,_,_,Flag,NewPiSet):-
    otherwise |
    NewPiSet := PiSet, Flag := false.
```

The analysis correctly infers that a call to `compose` will not suspend if its first, second and third arguments are ground. The predicate `lhs_strip_DmTm`, however, includes a debugging/error handling clause that merely contains a call to the pretty printer builtin `pp` (! flushes the output buffer):

```
lhs_strip_DmTm([],_,_,_,_):-
    pp('ERROR {Dm,Tm} not found in PiSet')!.
```

This clause is the source of the problem, since unlike the other clauses of `lhs_strip_DmTm`, it does ground its third, fourth and fifth arguments. In addition to printing a message, it is arguably better practise to abort the computation which can be accomplished by binding the output arguments to rogue values thereby causing the consuming `compose` processes to fail. Therefore this clause was modified to:

```
lhs_strip_DmTm([],_,C,D,E):-
    pp('ERROR Dm,Tm not found in PiSet')!,
    C := error, D := error, E := error.
```

Note that the problem was diagnosed by noting a mismatch between its success patterns of `lhs_strip_DmTm` and the call patterns of `compose`. In fact, to locate the source of a suspension, the programmer needs to consider the call graph structure and reason about success patterns and the inferred call patterns; all three pieces of information are required for effective suspension analysis on a larger application.

The analysis located similar error handling issues in the predicates `rhs_strip_DmTm`, `set_rf5` and `find_Dk_rf6` and Fisher confirmed these anomalies to be genuine bugs. The program was corrected, and when rerun with these modifications, the analysis successfully inferred that a call to `set_rf4` will not suspend if its first and third arguments are ground. The number of reruns of the analysis that were needed to debug `set_rf4` alone suggest that any practical system needs to be able to reanalyse a program within minutes; otherwise the cost of analysis will be prohibitively high for interactive debugging.

A call pattern of false was then inferred for `collect_matches` which occurs in SCC 176. This predicate contains 17 clauses all of which perform tuple manipulate. Rather than inspect the clauses manually, the analysis was repeatedly applied to isolate the source of the suspension. Initially all but one of the clauses were commented out. Then each clause was uncommented and the analysis rerun until a call pattern of false was inferred for collect_matches. This tactic revealed that the suspension arose from the following clause:

```
collect_matches(n,X,_,_,[d,Y|T],Acc,NSet):-
    dummy_match(d,Y,X,PiSet),
    next_collect(PiSet,n,X,_,_,T,Acc,NSet).
```

Note that this method of diagnosis that hinges on removing/reinserting clauses and then rerunning the analysis, seems key to debugging larger predicates.

The analysis inferred that a call to next_collect will not suspend if called with its first and second arguments ground. However, this particular clause invokes next_collect with the non-ground tuple {n,{X,_},_}. Inspection of next_collect (and those predicates it calls) revealed that these variables are never read or written. It is good programming practise to replace such variables with unique dummy terms, as illustrated below, to generate an error if the next_collect predicate (and those it calls) were incorrectly modified to read or write to these tuple arguments.

```
collect_matches({n,{X,_},_},[{d,Y}|T],Acc,NSet):-
    dummy_match({d,Y},X,PiSet),
    next_collect(PiSet,{n,{X, dummy}, dummy},T,Acc,NSet).
```

This modification is also sufficient for inferring that collect_matches will not suspend it is called with its first and second arguments ground.

Next, a call pattern of false was found in SCC 222 which contains the predicate distribute_tasks. The predicate distribute_tasks invokes another named produce_sets (via 4 intermediate calls) which includes a base clause which is problematic for analysis:

```
% Another call has dealt with the assignments.
produce_sets(_,_,_,_,_,_,_,_,true,_,_,_,_).
% Path closed elsewhere but no other call fired.
produce_sets(_,_,_,_,_,_,_,closed,false,
        TopPiSets,NSets,NewTasks,Tests):-
    NewTasks = [], TopPiSets = [[]],
    NSets = [[]], Tests = [closed].
```

Unlike the other clauses of this predicate, the first clause neither instantiates its final 4 arguments nor checks that they are completely instantiated. The clause seems to detect that another process has already instantiated these arguments. It is unlikely that any analysis will be able to trace this form of interaction and this predicate illustrates that although analysis can aid manual debugging, it cannot completely replace it. This suspension in turn induces two further suspensions in the predicates `solve` and `tabx` in the SCCs 226 and 227 (the two topmost SCCs) as a result of the suspension in `distribute_tasks`. Therefore the analysis infers non-trivial suspension conditions for all but 3 of the 319 predicates in the program.

9 Related work

The earliest work on suspension analysis [10] presents an and-or tree framework [3] 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 [8]. Finiteness is enforced through a widening known as star-abstraction [8]. This approach achieves a degree of conceptual simplicity though the abstract states themselves can be unwieldy. The work of [16] 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 [9]. 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 [20, 38] but none of these are geared towards goal-independence.

Surprisingly, one closely related work comes from the compiling control literature [4]; the work of [28] considers the problem of generating a local selection rule under which a program universally terminates. The technique of [28] builds on the termination inference method of [42] which infers initial modes for a query that, if satisfied, ensure that a logic program left-terminates. The chief advance in [28] over [42] 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 [42] 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 goal reorderings.

10 Conclusions

This paper has shown how suspension analysis can be tackled from a new perspective – that of goal-independence. The paper has argued that goal-independence is advantageous in that it enables the programmer to activate the analysis at the press of a button. Moreover, it leads to an analysis whose results and underlying conceptual model that can be comprehended by the programmer; the programmer can understand the suspension conditions inferred for a clause by studying the suspension conditions and success patterns inferred for the body atoms of that clause. Despite its generality, the goal-independent approach to suspension analysis can be realised efficiently even with relatively simple fixpoint engines. The efficiency and simplicity follow from exploiting an interaction between monotonic and positive Boolean functions. This result enables the analysis to avoid the complexity of enumerating various goal interleavings. In practical terms, significance of increased efficiency is that it allows the programmer to apply the analysis frequently within the program development cycle. In practise, for example, it enables to the source of a suspension to be straightforwardly located within predicate defined over many clauses. The problematic clause can be found in a semi-automatic way by first commenting out the clauses and then second individually uncommenting them and reapplying the analysis until the suspension occurs. Finally, the paper demonstrates that the goal-independent approach to suspension analysis can strike a good balance between tractability and precision. Even with domains that merely track groundness, it is possible to locate bugs that otherwise would have been missed.

Acknowledgements

EU working group 23677 funded Samir Genaim while he visited the University of Kent. The Royal Society Short Visits Scheme grant number 16385 funded Andy King while he visited the University of Verona. We thank Jacob Howe and Fred Mesnard for their valuable comments on an earlier version [22] of this work. We would also like to thank Bart Massey, Evan Tick, Robert Johnson and Matthew Huntbach for providing benchmarks. We also thank Saumya Debray, Michael Fisher and Ian Foster for, respectively, clarifying details of bessel, PTMddd and ssd.

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] R. Bagnara and P. Schachte. Factorizing Equivalent Variable Pairs in ROBDD-Based Implementations of Pos. In *Algebraic Methodology and Software Technology*, volume 1548 of *Lecture Notes in Computer Science*, pages 471–485. Springer-Verlag, 1998.

[3] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *The Journal of Logic Programming*, 10(1/2/3&4):91–124, 1991.

[4] M. Bruynooghe, D. De Schreye, and B. Krekels. Compiling Control. *The Journal of Logic Programming*, 6(1&2):135–162, 1989.

[5] M. Carlsson. Freeze, Indexing and Other Implementation Issues in the WAM. In *International Conference on Logic Programming*, pages 40–58. MIT Press, 1987.

[6] K. L. Clark, F. G. McCabe, and S. Gregory. IC-Prolog Language Features. In K. L. Clark and S.-Å. Tärnlund, editors, *Logic Programming*, pages 253–266. Academic Press, 1982.

[7] M. Codish and B. Demoen. Analyzing Logic Programs Using PROPositional Logic Programs and a Magic Wand. *The Journal of Logic Programming*, 25(3):249–274, 1995.

[8] M. Codish, M. Falaschi, and K. Marriott. Suspension Analyses for Concurrent Logic Programs. *ACM Transactions on Programming Languages and Systems*, 16(3):649–686, 1994.

[9] 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.

[10] 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.

[11] A. Cortesi, G. Filé, and W. H. Winsborough. Optimal Groundness Analysis Using Propositional Logic. *The Journal of Logic Programming*, 27(2):137–167, 1996.

[12] P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *The Journal of Logic Programming*, 13(2–3):103–179, 1992.

[13] V. Dahl. Two Solutions for the Negation Problem. In S.-Å. Tärnlund, editor, *Workshop on Logic Programming*, 1980.

[14] P. Dart. On Derived Dependencies and Connected Databases. *The Journal of Logic Programming*, 11(1–2):163–188, 1991.

[15] S. K. Debray. QD-Janus: a Sequential Implementation of Janus in Prolog. *Software, Practice and Experience*, 23(12):1337–1360, 1993.

[16] S. K. Debray, D. Gudeman, and P. Bigot. Detection and Optimization of Suspension-free Logic Programs. *The Journal of Logic Programming*, 29(1–3):171–194, 1996.

[17] G. Filé, R. Giacobazzi, and F. Ranzato. A Unifying View of Abstract Domain Design. *ACM Computing Surveys*, 28(2):333–336, 1996.

[18] G. Filé and F. Ranzato. The Powerset Operator on Abstract Interpretations. *Theoretical Computer Science*, 222(1–2):77–111, 1999.

[19] I. Foster and S. Taylor. *Strand: New Concepts in Parallel Programming.* Prentice-Hall, 1989.

[20] 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.

[21] S. Genaim and M. Codish. Inferring Termination Conditions for Logic Programs using Backwards Analysis. *Theory and Practice of Logic Programming*, 2005. http://xxx.soton.ac.uk/abs/cs.PL/0312023, To appear.

[22] S. Genaim and A. King. Goal-Independent Suspension Analysis for Logic Programs with Dynamic Scheduling. In P. Degano, editor, *European Symposium on Programming*, volume 2618 of *Lecture Notes in Computer Science*, pages 84–98. Springer-Verlag, 2003.

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

[24] R. Giacobazzi and F. Scozzari. A Logical Model for Relational Abstract Domains. *ACM Transactions on Programming Languages and Systems*, 20(5):1067–1109, 1998.

[25] D. C. Gras and M. V. Hermenegildo. Distributed WWW Programming using Ciao-Prolog and the PiLLoW library. *Theory and Practice of Logic Programming*, 1(3):251–282, 2001.

[26] S. Gregory. *Parallel Programming in Parlog*. Addison-Wesley, 1987.

[27] A. Heaton and A. King. Abstracting Builtins for Groundness Analysis. Technical Report 19-04, Computing Laboratory, University of Kent, CT2 7NF, 2000. http://www.cs.kent.ac.uk/pubs/2000/957.

[28] 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.

[29] J. M. Howe and A. King. Positive Boolean Functions as Multiheaded Clauses. In *International Conference on Logic Programming*, volume 2237 of *Lecture Notes in Computer Science*, pages 120–134. Springer-Verlag, 2001.

[30] M. M. Huntbach and G. A. Ringwood. *Agent-Oriented Programming*, volume 1630 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1999.

[31] J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The CLP($\mathcal{R}$) Language and System. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, 1992.

[32] R. Johnson. A Blackboard Approach to Parallel Temporal Tableaux. In P. Jorrand and V. Sgurev, editors, *Proceedings of the Sixth Inter-*

national Conference on Artificial Intelligence, Methodologies, Systems and Applications. World Scientific, 1994.

[33] A. King and L. Lu. A Backward Analysis for Constraint Logic Programs. *Theory and Practice of Logic Programming*, 2(4–5):517–547, 2002.

[34] R. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, 1979.

[35] J-L. Lassez, M. Maher, and K. Marriott. Unification Revisited. In *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann, 1988.

[36] B. Le Charlier and P. Van Hentenryck. Reexecution in Abstract Interpretation of Prolog. *Acta Informatica*, 32:209–253, 1995.

[37] S. Lüttringhaus-Kappel. Control Generation for Logic Programs. In D. S. Warren, editor, *International Conference on Logic Programming*, pages 478–495. The MIT Press, 1993.

[38] 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.

[39] K. Marriott and H. Søndergaard. Precise and Efficient Groundness Analysis for Logic Programs. *ACM Letters on Programming Languages and Systems*, 2(4):181–196, 1993.

[40] J. C. Martin and A. King. Control Generation by Program Transformation. Technical Report 5-04, Computing Laboratory, University of Kent, CT2 7NF, March 2004. http://www.cs.kent.ac.uk/pubs/2004/1868.

[41] C. Mellish. Abstract Interpretation of Prolog Programs. In *International Conference on Logic Programming*, volume 225 of *Lecture Notes in Computer Science*, pages 463–474. Springer-Verlag, 1986.

[42] 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.

[43] F. Mesnard and S. Ruggieri. On Proving Left Termination of Constraint Logic Programs. *ACM Transactions on Computational Logic*, 4(2):207–259, 2003.

[44] L. Naish. *Negation and Control in Logic Programs*. Springer-Verlag, 1986.

[45] L. Naish. Coroutining and the Construction of Terminating Logic Programs. *Australian Computer Science Communications*, 15(1):181–190, 1993.

[46] A. San Miguel Aguirre and M. Y. Vardi. Random 3-SAT and BDDs: The Plot Thickens Further. In T. Walsh, editor, *Principles and Practice of Constraint Programming*, volume 2239 of *Lecture Notes in Computer Science*, pages 121–136. Springer-Verlag, 2001.

[47] SICS. *SICStus Prolog User's Manual*, 2004. See http://www.sics.se/sicstus/.

[48] R. Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

[49] E. Tick. *Parallel Logic Programming*. MIT Press, 1991.

[50] E. Tick, B. C. Massey, and J. S. Larson. Experience with the Super Monaco Optimizing Compiler. *The Journal of Logic Programming*, 29(1–3):141–169, 1996.

[51] M. H. van Emden and G. J. de Lucena Filho. Predicate Logic as a Language for Parallel Programming. In K. L. Clark and S.-Å. Tärnlund, editors, *Logic Programming*, pages 189–198. Academic Press, 1982.

[52] T. Vasak and J. Potter. Characterisation of Terminating Logic Programs. In *Symposium on Logic Programming*, pages 140–147. IEEE Press, 1986.

[53] L. Vielle. Recursive Query Processing: The Power of Logic. *Theoretical Computer Science*, 69(1):1–53, 1989.

[54] J. Wunderwald. Memoing Evaluation by Source-to-Source Transformation. In *Proceedings of Logic Program Synthesis and Transformation*, volume 1048 of *Lecture Notes in Computer Sciene*, pages 17–32. Springer-Verlag, 1995.

A Proof Appendix

Proof A.1 (for proposition 5.4)

- *To show* $\forall_x f \models f$. *If* $\forall_x f = \text{false}$ *the result is immediate. Thus suppose* $\forall_x f = f[x \mapsto \text{true}] \wedge f[x \mapsto \text{false}]$. *Let* $M \in model_X(\forall_x f)$. *If* $x \in M$ *then* $M \in model_X(f)$ *since* $M \in model_X(f[x \mapsto \text{true}]) \supseteq model_X(\forall_x f)$. *Likewise if* $x \notin M$ *then* $M \in model_X(f)$ *since* $M \in model_X(f[x \mapsto \text{false}]) \supseteq model_X(\forall_x f)$.

- *Let* $g \models f$ *such that* $g \in Pos_{X \setminus \{x\}}$. *To show* $g \models \forall_x f$. *Let* $M \in model_X(g)$.

  - *Suppose* $x \in M$. *Because* $g \models f$, $M \in model_X(f)$ *and since* $x \in M$ *it follows that* $M \in model_X(f[x \mapsto \text{true}])$. *Because* $x \notin var(g)$, $M \setminus \{x\} \in model_X(g)$, *hence* $M \setminus \{x\} \in model_X(f)$ *and* $M \setminus \{x\} \in model_X(f[x \mapsto \text{false}])$. *Since* $x \notin var(f[x \mapsto \text{false}])$, $M \in model_X(f[x \mapsto \text{false}])$. *Thus* $M \in model_X(\forall_x f)$.
  - *Suppose* $x \notin M$. *Symmetric to the previous case.*

Therefore $\forall_x f = \vee\{g \in Pos_{X \setminus \{x\}} \mid g \models f\}$ *as required.*

Proof A.2 (for proposition 5.6) *Put* $Y = \{y \in X \mid f \models y\}$ *and* $g = \wedge Y$. *Then* $f = g \wedge (g \to f)$. *Since* $f \in Def_X$, $g \to f \in Def_X$, *hence* $g \to f = \wedge_{k \in K} y_k \leftarrow Y_k$ *for some (possibly empty) index set* K *where* $y_k \notin Y_k$ *and* $Y \subset Y_k$. *Thus* $y_k \notin Y$, *hence* $Y \in model_X(y_k \leftarrow Y_k)$ *and therefore* $Y \in model_X(g \to f)$. *Moreover,* $Y \in model_X(g)$ *so that* $Y \in model_X(f)$. *Since* $f \not\models d_i$ *and* $f \models g$, *it follows* $g \not\models d_i$. *Because* $d_i \in Mon_X$, $Y \notin model_X(d_i)$, *hence* $Y \in model_X(d_i \to f_i)$ *and thus* $Y \in model_X(f \wedge \wedge_{j=1}^m (d_j \to f_j))$. *For a contradiction, suppose there exists* $i \in [1, m]$ *such that* $f \wedge \wedge_{j=1}^m (d_j \to f_j) \models d_i$. *Then* $Y \in model_X(d_i)$ *and since* $d_i \in Mon_X$, *a contradiction* $f \models d_i$ *is obtained as required.*

Lemma A.3 *Let* $f \in Pos_X$ *and* $d = \wedge\{x \to \rho(x) \mid x \in X\}$.

- *Let* $g \in Mon_X$. *Then* $g \models f$ *iff* $g \wedge d \models \rho(f)$.
- *Let* $g \in Pos_X$. *If* $g \wedge d \models \rho(f)$ *then* $(\uparrow g) \wedge d \models \rho(f)$.

Proof A.4 (for lemma A.3) *Put* $Y = X \cup \rho(X)$.

- *Let* $g \in Mon_X$. *To show* $g \models f$ *iff* $g \wedge d \models \rho(f)$.

- *Suppose* $g \wedge d \models \rho(f)$. *Let* $M \in model_X(g)$. *Then* $M \cup \rho(M) \in model_Y(d)$, *hence* $M \cup \rho(M) \in model_Y(g \wedge d)$, *thus* $M \cup \rho(M) \in model_Y(\rho(f))$. *Therefore* $M \in model_X(f)$.
- *Suppose* $g \models f$. *Let* $M_1 \cup M_2 \in model_Y(g \wedge d)$ *where* $M_1 \subseteq X$ *and* $M_2 \subseteq \rho(X)$. *Put* $M_1' = \rho^{-1}(M_2) \supseteq M_1$. *Because* $g \in Mon_X$, $M_1' \in model_X(g)$ *therefore* $M_1' \in model_X(f)$ *thus* $M_2 \in model_X(\rho(f))$. *Hence* $M_1 \cup M_2 \in model_Y(\rho(f))$.

- *Let* $g \in Pos_X$ *and suppose* $g \wedge d \models \rho(f)$. *Let* $M_1 \cup M_2 \in model_Y((\uparrow g) \wedge d)$ *where* $M_1 \subseteq X$ *and* $M_2 \subseteq \rho(X)$. *Since* $\uparrow g \in Mon_X$ *there exists* $M_1' \subseteq M_1$ *such that* $M_1' \in model_Y(g)$. *But* $M_1' \cup M_2 \in model_Y(g \wedge d)$ *hence* $M_1' \cup M_2 \in model_Y(\rho(f))$ *thus* $M_2 \in model_{\rho(X)}(\rho(f))$ *and therefore* $M_1 \cup M_2 \in model_Y(\rho(f))$.

Proof A.5 *Observe* $(d \to \rho(f)) \wedge d \models \rho(f)$. *By lemma A.3*, $\uparrow(d \to \rho(f)) \wedge d \models \rho(f)$. *Thus* $\uparrow(d \to \rho(f)) \models d \to \rho(f)$ *and hence* $d \to \rho(f) \in Mon_{X \cup \rho(X)}$. *By corollary 5.5*, $\forall_{X'}(d \to \rho(f)) = \vee\{g \in Mon_X \mid g \models d \to \rho(f)\}$. *Let* $g \in Mon_X$. *Moreover* $g \models d \to \rho(f)$ *iff* $g \wedge d \models \rho(f)$ *iff* $g \models f$ *by lemma A.3. Therefore* $\forall_{X'}(d \to \rho(f)) = \vee\{g \in Mon_X \mid g \models f\}$.

Proof A.6 (for proposition 6.7) *Suppose* $f_1, f_2 \in Mon_{var(\vec{x})}$ *such that for all* $\theta \in \gamma_{var(\vec{x})}(f_i)$ *the predicate* $select(p(\vec{x}), \theta)$ *holds. Since* $\gamma_{var(\vec{x})}(f_1 \vee f_2) = \gamma_{var(\vec{x})}(f_1) \cup \gamma_{var(\vec{x})}(f_2)$ *it follows that* $select(p(\vec{x}), \theta)$ *holds for all* $\theta \in \gamma_{var(\vec{x})}(f_1 \vee f_2)$.

Proof A.7 (for theorem 6.13) Let $\langle p(\vec{z}), \theta\rangle \twoheadrightarrow_{local}{}^{l} susp$. Proof by induction on k.

- Suppose $\langle p(\vec{z}), \theta\rangle \in \gamma(\mathcal{B}'(Base'))$. Thus there exists $\langle p(\vec{z}), d'\rangle \in \mathcal{B}'^{k+1}(Base')$ such that $\alpha(\theta) \models d'$. Moreover $d' \models select'(p(\vec{z}))$, hence $\theta \in \gamma_{\vec{z}}(select'(p(\vec{z})))$, thus $select(p(\vec{z}), \theta)$ holds, therefore $l \geq 1$ as required.

- Suppose $\langle p(\vec{z}), \theta\rangle \in \gamma(\mathcal{B}'^{k+1}(Base'))$. Thus there exists $\langle p(\vec{z}), d'\rangle \in \mathcal{B}'^{k+1}(Base')$ such that $\alpha(\theta) \models d'$. Moreover $d' \models select'(p(\vec{z}))$, hence $\theta \in \gamma_{\vec{z}}(select'(p(\vec{z})))$, thus $select(p(\vec{z}), \theta)$ holds, whence $\langle p(\vec{z}), \theta\rangle \twoheadrightarrow_{local}{}^{1} \langle g, \delta \circ \theta\rangle$ where $\rho \in Ren$, $c = p(\vec{y})$:- $e, g \in \rho(P)$, $\mathrm{var}(c) \cap \mathrm{var}(\langle\{p(\vec{z})\}, \theta\rangle) = \emptyset$ and $\delta \in imgu(\{\theta(\vec{z}) = \vec{y}\} \cup e)$. Hence $\langle g, \delta \circ \theta\rangle \twoheadrightarrow_{local}{}^{l-1} susp$.

  Let $g = p_1(\vec{y}_1), \ldots, p_n(\vec{y}_n)$ and put $\psi_0 = \delta \circ \theta$. Let $\pi : [1, n] \to [1, n]$ be a permutation and $m \in [0, n-1]$ such that $\langle p_{\pi(i)}(\vec{y}_{\pi(i)}), \psi_{i-1}\rangle \twoheadrightarrow_{local}{}^{l_i} \langle \epsilon, \psi_i\rangle$ for each $i \in [1, m]$ and $\langle\{p_{\pi(i)}(\vec{y}_{\pi(i)}) \mid i \in [m+1, n]\}, \psi_m\rangle \twoheadrightarrow_{local}{}^{l-(1+\sum_{i=1}^{m} l_i)} susp$. By theorem 4.7 it follows that $\langle p_{\pi(i)}(\vec{y}_{\pi(i)}), \psi_i\rangle \in \mathcal{F}(P)$ for each $i \in$

$[1,m]$. By proposition 6.11 there exists $\langle p(\vec{y}_i), g_i\rangle \in \mathcal{F}'(P)$ such that $\alpha(\psi_i) \models g_i$ for each $i \in [1,m]$.

Since $\langle p(\vec{z}), d'\rangle \in \mathcal{B}'^{k+1}(Base')$ it follows $\langle p(\vec{y}), \exists_{\vec{z}}(\vec{z} \leftrightarrow \vec{y} \wedge d')\rangle \in \mathcal{B}'^{k+1}(Base')$. Therefore $\vec{z} \leftrightarrow \vec{y} \wedge d' \models \exists_{\vec{z}}(\vec{z} \leftrightarrow \vec{y} \wedge d') \models (\alpha(e) \wedge (\wedge_{i=1}^n d_i \rightarrow g_i)) \rightarrow (\wedge_{i=1}^n d_i)$ where $\langle p(\vec{y}_i), d_i\rangle \in \mathcal{B}'^k(Base')$. It follows that $\vec{z} \leftrightarrow \vec{y} \wedge d' \wedge \alpha(e) \models (\wedge_{i=1}^n d_i \rightarrow g_i) \rightarrow (\wedge_{i=1}^n d_i)$. But $\delta \in imgu(\{\theta(\vec{z}) = \vec{y}\} \cup e)$, therefore $\alpha(\delta \circ \theta) \models \alpha(\theta) \wedge (\vec{z} \leftrightarrow \vec{y}) \wedge \alpha(e) \models d' \wedge (\vec{z} \leftrightarrow \vec{y}) \wedge \alpha(e) \models (\wedge_{i=1}^n d_i \rightarrow g_i) \rightarrow (\wedge_{i=1}^n d_i)$.

Since $\psi_i \leq \psi_m$ for all $i \in [0,m]$ it follows $\alpha(\psi_m) \models \wedge_{i=1}^m g_{\pi(i)} \models \wedge_{i=1}^m d_{\pi(i)} \rightarrow g_{\pi(i)}$. Because $\alpha(\psi_m) \models \alpha(\delta \circ \theta)$, $\alpha(\psi_m) \models (\wedge_{i=m+1}^n d_{\pi(i)} \rightarrow g_{\pi(i)}) \rightarrow (\wedge_{i=1}^n d_\pi(i)) \models (\wedge_{i=m+1}^n d_{\pi(i)} \rightarrow g_{\pi(i)}) \rightarrow (\wedge_{i=m+1}^n d_{\pi(i)})$. Hence, by proposition 5.6, there exists $j \in [m+1,n]$ such that $\alpha(\psi_m) \models d_\pi(j)$. By the inductive hypothesis it follows that if $\langle p_{\pi(j)}(\vec{x}_{\pi(j)}), \psi_m\rangle \twoheadrightarrow_{local}{}^{l''} susp$ then $l'' \geq k$. Hence $l - 1 \geq l - (1 + \sum_{i=1}^m l_i) \geq l'' \geq k$ and therefore $l \geq k+1$ as required.

Note that proofs for standard and the particularly straightforwardly results are omitted for brevity.