Analysing Logic Programs by Reasoning Backwards

Jacob M. Howe[1], Andy King[2], and Lunjin Lu[3]

[1] City University, London, EC1V 0HB, UK
[2] University of Kent, Canterbury, CT2 7NF, UK
[3] Oakland University, Rochester, MI 48309, USA

Abstract. One recent advance in program development has been the application of abstract interpretation to verify the partial correctness of a (constraint) logic program. Traditionally forwards analysis has been applied that starts with an initial goal and traces the execution in the direction of the control-flow to approximate the program state at each program point. This is often enough to verify assertions that a property holds. The dual approach is to apply backwards analysis to propagate properties of the allowable states against the control-flow to infer queries for which the program will not violate any assertion. Backwards analysis also underpins other program development tasks such as verifying the termination of a logic program or proving that a logic program with a delay mechanism cannot reduce to a state that contains sub-goals which suspend indefinitely. This paper reviews various backwards analyses that have been proposed for logic programs, identifying common threads in these techniques. The analyses are explained through a series of worked examples. The paper concludes with some suggestions for research in backwards analysis for logic program development.

1 Introduction

Abstract interpretation has an important rôle in program development and specifically the verification and debugging of (constraint) logic programs, as recently demonstrated in [12,42,57]. In this context, programmers are typically equipped with an annotation language in which they encode properties of the program state at various program points [56,64]. One approach to verification of logic programs is to trace the program state in the direction of control-flow from an initial goal (forwards analysis), using abstract interpretation to finitely represent and track the state. The program is deemed to be correct if all the assertions are satisfied whenever they are encountered during the execution of the program; otherwise the program is potentially buggy. The dual approach is to trace execution against the control-flow (backwards analysis) to infer those queries which ensure that the assertions are satisfied should they be encountered [39,44]. If the class of initial queries does not conform to those expected by the programmer, then the program is potentially buggy.

Many program properties cannot be simply verified by checking (an abstraction of) the program store – they are properties of sequences of program states. For example, termination checking of logic programs attempts to verify that a logic program left-terminates for a given query [10,53]. This amounts to checking that sequences of program states are necessarily finite. Suspension analysis of concurrent logic programs [8,11,17] also reasons about sequences of states. It aims to verify that a given state cannot lead to another which possesses a sub-goal that suspends indefinitely. These classic analyses are inherently forwards since they trace sequences of states in the direction of control-flow. Nevertheless these forwards analysis problems (and various related analyses) have corresponding backwards analysis problems, tracing requirements against the control-flow (specifying the backwards analysis will be referred to as reversal). The reversal of termination checking is termination inference which infers initial queries under which a logic program left-terminates [24,53]. The reversal of suspension analysis is suspension inference [26] which infers a class of goals that will not lead to suspended (floundering) sub-goals. It has been observed [24] that the "missing link" between termination inference and termination checking is the backwards analysis of [39]. Likewise suspension inference [26] relies on ideas inherited from backwards analysis [39].

The unifying idea behind these various backwards analyses [24,26,39,40,44] is reasoning about reverse information flow. In abstract interpretation, information is represented, albeit in an approximate way, with an abstract domain which is a lattice $\langle D, \trianglelefteq, \oplus, \otimes \rangle$. The ordering $\trianglelefteq$ expresses the relative precision of two domain elements; the join $\oplus$ models the merging of computation paths whereas meet $\otimes$ models the conjunction of constraints. To propagate information against the control-flow, the analyses of [24,26,39,40] (but notably not that of [44]) require the abstract domain D to be relatively pseudo-complemented, that is, the relative pseudo-complement of two domain elements uniquely exists. The pseudo-complement of d_1 relative to d_2, denoted $d_1 \to d_2$, delivers the weakest element of D whose conjunction with d_1 implies d_2, or more exactly, $d_1 \to d_2 = \oplus\{d \in D \mid d \otimes d_1 \trianglelefteq d_2\}$. The rôle of relative pseudo-complement is that if d_2 expresses a set of requirements that must hold after a constraint is added to the store, and d_1 models the constraint itself, then $d_1 \to d_2$ expresses the requirements that must hold on the store before the constraint. Relative pseudo-complement is central to many backwards analyses.

Not all domains come equipped with a relative pseudo-complement, but it turns out that it is always possible to synthesise a domain for backwards analysis for some given downward closed property by applying Heyting completion [30]. Heyting completion enriches a domain with new elements so that the relative pseudo-complement is well-defined. All domains that are condensing possess a relative pseudo-complement. Examples of condensing domains include the class of positive Boolean functions [37,46], the relational type domain of [9], and the domain of directional types [1,30]. The requirement for a domain to be relatively pseudo-complemented is one the major restrictions of backwards analysis. Despite this limitation, backwards analysis still offers two key advantages over forwards analysis for program development problems. These advantages are summarised below:

- Backwards analysis generalises forwards analysis in that a single application of backwards analysis can subsume many applications of a forwards analysis. Another advantage that relates to ease of use is that backwards analysis is not dependent on the programmer (or the module system) for providing a top-level goal. This is because backwards analysis is goal-independent.
- In terms of implementation, backwards analysis strikes a good balance between generality and simplicity. Moreover, backwards analysis does not necessarily incur a performance penalty over forwards analysis.

Both of these two points are multi-faceted and therefore they require some unpacking. Returning to the first point – the issue of generality – forwards analysis is driven from a top-level goal. Forwards analysis then verifies that the goal (and those goals it recursively calls) satisfy a set of requirements. Conversely, backwards analysis infers a class of goals all of which are guaranteed to satisfy the requirements. Under certain algebraic conditions this class is maximal with respect to forwards analysis [40]; it describes all those goals that can be verified with forwards analysis. For example, consider the problem of understanding how to re-use code developed by a third party. In the context of logic programming, part of this problem reduces to figuring out how to query a program. If the logic program does not come with any documentation, then the programmer is forced to experiment with queries in an *ad hoc* fashion. More systematically, forwards analysis could be repeatedly applied to discover queries which are consistent with the called builtins in that the calls do not generate any instantiation errors. By way of contrast the backwards analysis framework when instantiated with a domain for tracking groundness dependencies [37,46] yields an analysis for mode inference which would discover (in a single application) all queries that will not generate any instantiation errors. This recovered mode information then provides valuable insight into behaviour of the program.

Expanding the second point – the issue of implementation – the analyses presented in [26,39,40] reduce to two simple bottom-up fixpoint computations: a least fixpoint (lfp) and a greatest fixpoint (gfp). The lfp and the gfp calculations can be ordered and thus de-coupled. This significantly simplifies the tracking of dependencies which is the main source of complexity in an efficient fixpoint engine. Moreover, although few forwards analyses have been compared experimentally against backwards analyses, the notable exception is in termination inference. In this context, the speed of inference appears to at least match that of checking [24]. In fact the total analysis time for checking and inference can be broken down into *Joint* – the time spent on activities common to both checking and inference and *Inf* and *Check* – the time spent on activities specific to inference and checking respectively. *Joint* dominates both *Inf* and *Check* but *Inf* is typically smaller than *Check* [24]. Therefore the generality of inference does not necessarily incur a performance penalty.

Backwards analysis has been applied extensively in functional programming in, among other things, projection analysis [65], stream strictness analysis [31], inverse image analysis [18]. Furthermore, backwards reasoning on imperative programs dates back to the early days of static analysis [14]. In contrast, backwards analysis has until very recently [19,22,24,26,40,44,49] been rarely applied in logic programming. The aim of this paper is thus to promote the use of backwards analysis especially within the context of logic program development. To this end, the paper explains the key ideas behind backwards analyses for mode inference, termination inference, suspension inference and type inference. These analyses are each described in an informal way through a series of worked examples in sections 2, 3, 4 and 5 respectively. Each of these sections includes its own related work section. Section 6 then reviews directions for future work on backwards analysis for program development and section 7 concludes.

2 Backwards mode inference

The objective of backwards analysis is to infer queries for which the program is guaranteed to either not violate any assertion or satisfy some operational requirement. To realise this objective, backwards analyses propagate requirements of the allowable states against the control-flow. This tactic essentially reinterprets the calculation of weakest pre-conditions [35] for logic programming using abstract interpretation techniques. To illustrate these ideas, consider the problem of mode inference [39]. In mode inference, the problem is to deduce moding properties which, if satisfied by the initial query, ensure that the resulting derivations cannot encounter an instantiation error. Instantiation errors arise when a builtin is called with insufficiently instantiated arguments. For example, the Prolog builtins `tab` and `put` require their first (and only) argument to be bound to a ground term otherwise they error. Conversely, the builtin `is` requires its last argument to be ground. Other builtins such as the arithmetic tests `=:=`, `<`, `>`, etc require both arguments to be ground. These grounding requirements can be expressed with the domain of positive Boolean functions, Pos, which is traditionally used to track groundness dependencies [37,46]. Pos is the set of functions $f : \{true, false\}^n \to \{true, false\}$ such that $f(true, \ldots, true) = true$. For example, $\mathrm{X} \wedge (\mathrm{Y} \leftarrow \mathrm{Z}) \in Pos$ since $true \wedge (true \leftarrow true) = true$. The formula describes states in which X is ground and Y is ground whenever Z is ground. Observe that this grounding property is closed under instantiation: if $\mathrm{X} \wedge (\mathrm{Y} \leftarrow \mathrm{Z})$ describes the state of the store, then both X and $\mathrm{Y} \leftarrow \mathrm{Z}$ still hold whenever the store is conjoined with additional constraints. When augmented with $false$, Pos forms the lattice $\langle Pos, \models, \vee, \wedge \rangle$ where $\models$ denotes

```
qs([], S, S) :- true.
qs([M | Xs], S, T) :- pt(Xs, M, L, H), qs(L, S, [M | R]), qs(H, R, T).

pt([], _, [], []) :- true.
pt([X | Xs], M, [X | L], H) :- M ≤ X, pt(Xs, M, L, H).
pt([X | Xs], M, L, [X | H]) :- M > X, pt(Xs, M, L, H).
```

Fig. 1. `quicksort` program in expressed in Prolog

the entailment ordering, $\wedge$ is logical conjunction and $\vee$ is logical disjunction. The top and bottom elements of the lattice are *true* and *false* respectively.

The assertions that are used in mode inference are *Pos* abstractions that express grounding requirements which ensure that instantiation errors are avoided. Specifically, an assertion is added to the program for each call to a builtin. The assertion itself precedes the call [39]. It is important to appreciate that the assertions only codify sufficient conditions; necessary conditions for the absence of instantiation errors cannot always be expressed within *Pos*. For example, the assertion $\mathtt{X} \vee (\mathtt{Y} \wedge \mathtt{Z})$ describes states for which the builtin `functor(X, Y, Z)` will not produce an instantiation error. Observe, however, that the same call will not generate an instantiation error if X is bound to a non-variable (non-ground) term such as `[W|Ws]`, hence $\mathtt{X} \vee (\mathtt{Y} \wedge \mathtt{Z})$ is not a *necessary* condition for avoiding an instantiation error. Observe too that $\mathtt{X} \vee (\mathtt{Y} \wedge \mathtt{Z})$ only ensures that the call will not generate an *instantiation* error. For instance, a *domain* error will be thrown whenever `functor(X, Y, Z)` is called with `Z` is instantiated to a negative integer. However a richer domain, such as the numeric power domain introduced in [40], could express this positivity requirement on the variable `Z`. (The subtlety of reasoning about builtins is not confined to backwards analysis. In fact correctly and precisely encoding the behaviour of the builtins is often the most difficult part of any analysis [33,36].)

2.1 Worked example on mode inference

To appreciate how the assertions and lattice operations $\vee$ and $\wedge$ fit together and why the domain is required to be relatively pseudo-complemented, it is helpful to consider a worked example. Thus consider the quicksort program listed in Figure 1 and the problem of computing those queries that avoid instantiation errors. The quicksort program is coded in Prolog and therefore the comma operator denotes sequential (rather than parallel) goal composition. A difference list is used to amortise the cost of appending the two lists produced by the goals `qs(L, S, [M | R])` and `qs(H, R, T)`.

The backwards analysis consists of two computational steps. The first is a least fixpoint (lfp) calculation and the second is a greatest fixpoint (gfp) computation. The lfp is an analysis on its own right. It infers success patterns that are required for the gfp computation. The success pattern for a given predicate characterises the bindings made by the predicate whenever it succeeds; in this context the success patterns are described as groundness dependencies. Specifically, the lfp is a set of calls paired with groundness dependencies which describe how a call to each predicate in the program can succeed. The gfp is an analysis for input modes (the objective of the backwards analysis). To simplify both steps, the program is put into a form in which the arguments of head and body atoms are distinct variables. This gives the normalised program listed in the first column of Figure 2. To clearly differentiate assertions from the (Herbrand) constraints that occur within the program, the program is expressed in the concurrent constraint style [60] using ask to denote an assertion and tell to indicate a conventional store write. This notation (correctly) suggests that an assertion reads and checks a property of the store. Empty conjunctions of atoms are denoted by `true`. The process of normalisation does not introduce any assertions and therefore the program in the first column of Figure 2 includes only tell constraints. Note that each clause contains a single tell constraint which appears immediately before the (normalised) atoms that constitute the body of the clause.

After normalisation, the program is abstracted by replacing each tell constraint $x = f(x_1, \ldots, x_n)$ with a formula $x \leftrightarrow \wedge_{i=1}^{n} x_i$ that describes its groundness dependencies. This gives the abstract program listed in the second column of Figure 2. Builtins that are called from the program, such as the tests $\leq$ and $>$, are handled by augmenting the abstract program with fresh predicates, $\leq'$ and $>'$, which capture the grounding behaviour of the builtins. Assertions are introduced immediately after the head of these fresh clauses which specify a mode that is sufficient for the builtin not to generate an instantiation error. For example, the ask formula in the $\leq'$ clause asserts that the $\leq$ test will not error if its first two arguments are ground, whereas the tell formula describes the state that holds if the test succeeds. For uniformity, all clauses contain both an ask and a tell. This normal form simplifies the presentation of the theory and well as the structure of the abstract interpretation itself. In practise, the ask of most clauses are `true` and thus vacuous. In the case of quicksort, the only non-trivial assertions arise from builtins. This would change if the programmer introduced assertions for purposes of verification [56].

2.2 Least fixpoint calculation

An iterative algorithm is used to compute the lfp and thereby characterise the success patterns of the program. A success pattern is a pair consisting of an atom with distinct variables for arguments paired with a *Pos* formula over those variables which describes the groundness dependencies between the arguments. Renaming and equality of formulae induce an equivalence between success patterns which is needed to detect the fixpoint. The patterns

```
qs(T1, S, T2) :-                    qs(T1, S, T2) :-
   tell(T1 = [], T2 = S),              ask(true),
   true.                               tell(T1 ∧ (T2 ↔ S),
qs(T1, S, T3) :-                       true.
   tell(T1 = [M|Xs], T3 = [M|R]),   qs(T1, S, T3) :-
   pt(Xs, M, L, H),                    ask(true),
   qs(L, S, T3),                       tell(T1 ↔ (M ∧ Xs) ∧ T3 ↔ (M ∧ R)),
   qs(H, R, T).                        pt(Xs, M, L, H),
                                       qs(L, S, T3),
pt(T1, _, T2, T3) :-                   qs(H, R, T).
   tell(T1=[], T2=[], T3=[]),
   true.                            pt(T1, _, T2, T3) :-
pt(T1, M, T2, H) :-                    ask(true),
   tell(T1 = [X|Xs], T2 = [X|L]),      tell(T1 ∧ T2 ∧ T3),
   M ≤ X,                              true.
   pt(Xs, M, L, H).                 pt(T1, M, T2, H) :-
pt(T1, M, l, T2) :-                    ask(true),
   tell(T1 = [X|Xs], T2 = [X|H]),      tell(T1 ↔ (X ∧ Xs) ∧ T2 ↔ (X ∧ L)),
   M > X,                              ≤'(M, X),
   pt(Xs, M, L, H).                    pt(Xs, M, L, H).
                                    pt(T1, M, L, T2) :-
                                       ask(true),
                                       tell(T1 ↔ (X ∧ Xs) ∧ T2 ↔ (X ∧ H)),
                                       >'(M, X),
                                       pt(Xs, M, L, H).

                                    ≤'(M, X) :-
                                       ask(M ∧ X), tell(M ∧ X), true.

                                    >'(M, X) :-
                                       ask(M ∧ X), tell(M ∧ X), true.
```

Fig. 2. quicksort program with assertions and as a *Pos* abstraction

$\langle p(u, w, v), u \wedge (w \leftrightarrow v)\rangle$ and $\langle p(x_1, x_2, x_3), (x_3 \leftrightarrow x_2) \wedge x_1\rangle$, for example, are considered to be identical: both express the same inter-argument groundness dependencies. Each iteration produces a set of success patterns: at most one pair for each predicate in the program.

Upper approximation of success patterns A success pattern records an inter-argument groundness dependency that describes the binding effects of executing a predicate. If $\langle p(\boldsymbol{x}), f\rangle$ correctly describes the predicate p, and g holds whenever f holds, then $\langle p(\boldsymbol{x}), g\rangle$ also correctly describes p. Note that here and henceforth $\boldsymbol{x}$ denotes a vector of distinct variables. Success patterns can thus be approximated from *above* without compromising correctness. Iteration is performed in a bottom-up fashion, T_P-style, [28] and commences with $F_0 = \{\langle p(\boldsymbol{x}), false\rangle \mid p \in \Pi\}$ where Π is the set of predicates occurring in the program. F_0 is the bottom element of the lattice of success patterns; the top element is $\{\langle p(\boldsymbol{x}), true\rangle \mid p \in \Pi\}$. F_{j+1} is computed from F_j by considering each clause $p(\boldsymbol{x})$:- $\mathsf{ask}(d), \mathsf{tell}(f), p_1(\boldsymbol{x}_1), \ldots, p_n(\boldsymbol{x}_n)$ in turn. It is at this stage that the lattice structure of Pos comes into play. Meet (the operator $\wedge$ which is also known as greatest lower bound) provides a way of conjoining information from different body atoms, while join (the operator $\vee$ which is also known as least upper bound) is used to combine the information from different clauses. More exactly, the success pattern formulae f_i for the n body atoms $p_1(\boldsymbol{x}_1), \ldots, p_n(\boldsymbol{x}_n)$ are conjoined with f to obtain $g = f \wedge (\wedge_{i=1}^n f_i)$. Variables not present in $p(\boldsymbol{x})$, Y say, are then eliminated from g. The Schröder elimination principle provides a way of eliminating a variable from a given formula. It enables a projection operator $\exists_x$ to be defined by $\exists_x(f) = f[x \mapsto true] \vee f[x \mapsto false]$ which eliminates x from f. Since $f \models \exists_x(f)$, computing $g' = \exists_Y(g)$ where $\exists_{\{y_1 \ldots y_n\}}(g) = \exists_{y_1}(\ldots \exists_{y_n}(g))$ weakens g. Weakening g does not compromise correctness because success patterns can be safely approximated from above.

Weakening upper approximations The pattern $\langle p(\boldsymbol{x}), g''\rangle$ where g'' is the current Pos abstraction is then replaced with $\langle p(\boldsymbol{x}), g' \vee g''\rangle$ where g' is computed as above. Thus the success patterns become progressively weaker (or at least not stronger) on each iteration. Again, correctness is preserved because success patterns can be safely approximated from above.

Least fixpoint calculation for quicksort The lfp for the abstracted quicksort program is obtained (and checked) in the following 3 iterations:

$$F_1 = \left\{ \begin{array}{c} \langle \mathtt{qs}(x_1, x_2, x_3), x_1 \wedge (x_2 \leftrightarrow x_3)\rangle \\ \langle \mathtt{pt}(x_1, x_2, x_3, x_4), x_1 \wedge x_3 \wedge x_4\rangle \\ \langle {=<}'(x_1, x_2), x_1 \wedge x_2\rangle \\ \langle {>}'(x_1, x_2), x_1 \wedge x_2\rangle \end{array} \right\}$$

$$F_2 = \left\{ \begin{array}{c} \langle \mathtt{qs}(x_1, x_2, x_3), x_2 \leftrightarrow (x_1 \wedge x_3)\rangle \\ \langle \mathtt{pt}(x_1, x_2, x_3, x_4), x_1 \wedge x_3 \wedge x_4\rangle \\ \langle {=<}'(x_1, x_2), x_1 \wedge x_2\rangle \\ \langle {>}'(x_1, x_2), x_1 \wedge x_2\rangle \end{array} \right\}$$

Finally, $F_3 = F_2$. The space of success patterns forms a complete lattice which ensures that a lfp exists. The iterative process will always terminate since the space is finite and hence the number of times each success pattern can be updated is also finite. Moreover, it will converge onto the lfp since (so-called Kleene) iteration commences with the bottom element F_0.

Observe that F_2, the lfp, faithfully describes the grounding behaviour of quicksort: a `qs` goal will ground its second argument if it is called with its first and third arguments already ground and *vice versa*. Note that assertions are not considered in the lfp calculation.

2.3 Greatest fixpoint calculation

A bottom-up strategy is used to compute a gfp and thereby characterise the safe call patterns of the program. A safe call pattern describes queries that do not lead to violation of the assertions. A call pattern has the same form as a success pattern (so there is one call pattern per predicate rather than one per clause). The analysis starts by checking that no call causes an error by reasoning backwards over all clauses. If an assertion is violated, the set of safe call patterns for the involved predicate is strengthened (made smaller), and the whole process is repeated until the assumptions turn out to be valid (the gfp is reached).

Lower approximation of safe call patterns Iteration commences with the top element $D_0 = \{\langle p(\boldsymbol{x}), true\rangle \mid p \in \Pi\}$. An iterative algorithm incrementally *strengthens* the call pattern formulae until they only describe queries which lead to computations that satisfy the assertions. Note that call patterns describe a subset, rather than a superset, of those queries which are safe. Call patterns are thus lower approximations, in contrast to success patterns which are upper approximations. Put another way, if $\langle p(\boldsymbol{x}), g\rangle$ correctly describes some safe call patterns of p, and g holds whenever f holds, then $\langle p(\boldsymbol{x}), f\rangle$ also correctly describes some safe call patterns of p. Call patterns can thus be approximated from *below* without compromising correctness (but not from above). D_{k+1} is computed from D_k by applying each $p(\boldsymbol{x})$:- $\mathsf{ask}(d), \mathsf{tell}(f), p_1(\boldsymbol{x}_1), \ldots, p_n(\boldsymbol{x}_n)$ in turn and calculating a formula that characterises its safe calling modes. A safe calling mode is calculated by propagating moding requirements right-to-left by repeated application of the logical operator $\rightarrow$. More exactly, let f_i denote the success pattern formula for $p_i(\boldsymbol{x}_i)$ in the previously computed lfp and let d_i denote the call pattern formula for $p_i(\boldsymbol{x}_i)$ in D_k. Set $e_{n+1} = true$ and then compute $e_i = d_i \wedge (f_i \rightarrow e_{i+1})$ for $1 \leq i \leq n$. Each e_i describes a safe calling mode for the compound goal $p_i(\boldsymbol{x}_i), \ldots, p_n(\boldsymbol{x}_n)$.

Intuition and explanation The intuition behind the symbolism is that d_i represents the demand that is already known in order for $p_i(\boldsymbol{x}_i)$ not to error whereas e_i is d_i possibly strengthened with extra demand so as to ensure that the sub-goal $p_{i+1}(\boldsymbol{x}_{i+1}), \ldots, p_n(\boldsymbol{x}_n)$ also does not error when executed immediately after $p_i(\boldsymbol{x}_i)$. Put another way, anything larger than d_i may possibly cause an error when executing $p_i(\boldsymbol{x}_i)$ and anything larger than e_i may possibly cause an error when executing $p_i(\boldsymbol{x}_i), \ldots, p_n(\boldsymbol{x}_n)$.

The basic inductive step in the analysis is to compute an e_i which ensures that $p_i(\boldsymbol{x}_i), \ldots, p_n(\boldsymbol{x}_n)$ does not error, given d_i and e_{i+1} which respectively

```
less_than_one(X, Flag) :- X < 1, Flag = 1.
less_than_one(X, Flag) :- 1 =< X, Flag = 0.

less_than(X, Y) :- X < Y.
```

Fig. 3. The `less_than_one` and `less_than` predicates

ensure that $p_i(\boldsymbol{x}_i)$ and $p_{i+1}(\boldsymbol{x}_{i+1}), \ldots, p_n(\boldsymbol{x}_n)$ do not error. This step propagates a demand after the call to $p_i(\boldsymbol{x}_i)$ into a demand before the call to $p_i(\boldsymbol{x}_i)$. The tactic is to set $e_{n+1} = \mathit{true}$ and then compute $e_i = d_i \wedge (f_i \rightarrow e_{i+1})$ for $i \leq n$. This tactic is best explained by unfolding the definitions of e_n, then e_{n-1}, then e_{n-2}, and so on. This reverse ordering reflects the order in which the e_i are computed; the e_i are computed whilst walking backwards across the clause. Any calling mode is safe for the empty goal and hence $e_{n+1} = \mathit{true}$. Note that $e_n = d_n \wedge (f_n \rightarrow e_{n+1}) = d_n \wedge (\neg f_n \vee \mathit{true}) = d_n$. Hence e_n represents a safe calling mode for the goal $p_n(\boldsymbol{x}_n)$.

Observe that e_i should not be larger than d_i, otherwise an error may occur while executing $p_i(\boldsymbol{x}_i)$. Observe too that if $p_i(\boldsymbol{x}_i), \ldots, p_n(\boldsymbol{x}_n)$ is called with a mode described by d_i, then $p_{i+1}(\boldsymbol{x}_{i+1}), \ldots, p_n(\boldsymbol{x}_n)$ is called with a mode described by $(d_i \wedge f_i)$, since f_i describes the success patterns of $p_i(\boldsymbol{x}_i)$. The mode $(d_i \wedge f_i)$ may satisfy the e_{i+1} demand. If it does not, then the minimal extra demand is added to $(d_i \wedge f_i)$ so as to satisfy e_{i+1}. This minimal extra demand is $((d_i \wedge f_i) \rightarrow e_{i+1})$ – the *weakest* mode that, in conjunction with $(d_i \wedge f_i)$, ensures that e_{i+1} holds. Put another way, $((d_i \wedge f_i) \rightarrow e_{i+1}) = \vee\{f \in \mathit{Pos} \mid (d_i \wedge f_i) \wedge f \models e_{i+1}\}$. Combining the requirements to satisfy $p_i(\boldsymbol{x}_i)$ and then $p_{i+1}(\boldsymbol{x}_{i+1}), \ldots, p_n(\boldsymbol{x}_n)$, gives $e_i = d_i \wedge ((d_i \wedge f_i) \rightarrow e_{i+1})$ which reduces to $e_i = d_i \wedge (f_i \rightarrow e_{i+1})$ because of algebraic properties of condensing domains [30] and yields the tactic used in the basic inductive step.

To illustrate how requirements are combined for compound queries, consider the predicates `less_than_one` and `less_than` given in figure 3. The first predicate uses a flag to indicate whether its first argument is less than one; the second predicate is a test which succeeds if and only if its first argument is less than its second. In particular consider the (artificial) compound query `less_than_one(X, Flag), less_than(X, Flag)` which also succeeds whenever `X` is less than one. Observe that the query `less_than_one(X, Flag)` will not admit an instantiation error if the query is called with `X` sufficiently instantiated, that is, if `X` is ground. It is natural for this property also to hold for the compound query, since declaratively it encodes the same behaviour (albeit with some redundancy). However, reasoning about the instantiation requirements for `less_than_one(X, Flag), less_than(X, Flag)` is more subtle because the first sub-goal instantiates `Flag` thereby partially discharging the instantiation requirements of the second sub-goal. Moreover, the requirement that `X` is ground for the first sub-goal ensures that the same requirement is satisfied in the second sub-goal. Observe that this interaction is faithfully modelled by $e_i = d_i \wedge (f_i \rightarrow e_{i+1})$. Specifically, with $p_1(\boldsymbol{x}_1) = \texttt{less\_than\_one(X, Flag)}$ and $p_2(\boldsymbol{x}_2) = \texttt{less\_than(X, Flag)}$, the demand and success patterns for $p_1(\boldsymbol{x}_1)$ and $p_2(\boldsymbol{x}_2)$ are as follows $d_1 = \texttt{X}$ and $f_1 = \texttt{X} \wedge \texttt{Flag}$ and $d_2 = \texttt{X} \wedge \texttt{Flag}$ and $f_2 = \texttt{X} \wedge \texttt{Flag}$. Then

$$
\begin{aligned}
e_3 &= true \\
e_2 &= d_2 \wedge (f_2 \rightarrow e_3) = (\texttt{X} \wedge \texttt{Flag}) \wedge ((\texttt{X} \wedge \texttt{Flag}) \rightarrow true) = (\texttt{X} \wedge \texttt{Flag}) \\
e_1 &= d_1 \wedge (f_1 \rightarrow e_2) = (\texttt{X}) \wedge ((\texttt{X} \wedge \texttt{Flag}) \rightarrow (\texttt{X} \wedge \texttt{Flag})) = \texttt{X}
\end{aligned}
$$

Thus it is sufficient to ground `X` in order to avoid an instantiation error in the compound goal.

Pseudo-complement This step of calculating the *weakest* mode that when conjoined with $d_i \wedge f_i$ implies e_{i+1}, is the very heart of the analysis. Setting $e_i = false$ would trivially achieve safety, but e_i should be as weak as possible to maximise the class of safe queries inferred. For Pos, computing the weakest e_i reduces to applying the $\rightarrow$ operator, but more generally this step amounts to applying the relative pseudo-complement. This operation (if it exists for a given abstract domain) takes, as input, two abstractions and returns, as output, the *weakest* abstraction whose conjunction with the first input abstraction is at least as strong as the second input abstraction. If the domain does not possess a relative pseudo-complement, then there is not always a *unique* weakest abstraction (whose conjunction with one given abstraction is at least as strong as another given abstraction).

To see this, consider the domain Def [2,38] which does not possess a relative pseudo-complement. Def is the sub-class of Pos that is definite [2,38]. This means that Def has the special property that each of its Boolean functions can be expressed as a (possibly empty) conjunction of propositional Horn clauses. As with Pos, Def is assumed to be augmented with the bottom element $false$. Def can thus represent the groundness dependencies $x \wedge y$, x, $x \leftrightarrow y$, y, $x \leftarrow y$, $x \rightarrow y$, $false$ and $true$ but *not* $x \vee y$. Suppose that $d_i \wedge f_i = (x \leftrightarrow y)$ and $e_{i+1} = (x \wedge y)$. Then conjoining x with $d_i \wedge f_i$ would be at least as strong as e_{i+1} and symmetrically conjoining y with $d_i \wedge f_i$ would be at least as strong as e_{i+1}. However, Def does not contain a Boolean function strictly weaker than both x and y, namely $x \vee y$, whose conjunction with $d_i \wedge f_i$ is at least as strong as e_{i+1}. Thus setting $e_i = x$ or $e_i = y$ would be safe but setting $e_i = (x \vee y)$ is prohibited because $x \vee y$ falls outside Def. Moreover, setting $e_i = false$ would lose an unacceptable degree of precision. A choice would thus have to be made between setting $e_i = x$ and $e_i = y$ in some arbitrary fashion, so there would be no clear tactic for maximising precision.

Returning to the compound goal $p_i(\boldsymbol{x}_i), \ldots, p_n(\boldsymbol{x}_n)$, a call described by the mode $d_i \wedge ((d_i \wedge f_i) \rightarrow e_{i+1})$ is thus sufficient to ensure that neither $p_i(\boldsymbol{x}_i)$ nor the sub-goal $p_{i+1}(\boldsymbol{x}_{i+1}), \ldots, p_n(\boldsymbol{x}_n)$ error. Since $d_i \wedge ((d_i \wedge f_i) \rightarrow e_{i+1}) =$

$d_i \wedge (f_i \rightarrow e_{i+1}) = e_i$ it follows that $p_i(\boldsymbol{x}_i), \ldots, p_n(\boldsymbol{x}_n)$ will not error if its call is described by e_i. In particular, it follows that e_1 describes a safe calling mode for the body atoms of the clause $p(\boldsymbol{x})$:- $\mathsf{ask}(d), \mathsf{tell}(f), p_1(\boldsymbol{x}_1), \ldots, p_n(\boldsymbol{x}_n)$.

The next step is to calculate $g = d \wedge (f \rightarrow e_1)$. The abstraction f describes the grounding behaviour of the Herbrand constraint added to the store prior to executing the body atoms. Thus $(f \rightarrow e_1)$ describes the *weakest* mode that, in conjunction with f, ensures that e_1 holds, and hence the body atoms are called safely. Hence $d \wedge (f \rightarrow e_1)$ represents the weakest demand that both satisfies the body atoms and the assertion d. One subtlety which relates to the abstraction process is that d is required to be a lower-approximation of the assertion whereas f is required to be an upper-approximation of the constraint. Put another way, if the mode d describes the binding on the store, then the (concrete) assertion is satisfied, whereas if the (concrete) constraint is added to the store, then the store is described by the mode f.

Strengthening lower approximations The projection operator $\exists_x$ cannot be applied to eliminate variables in g that are not present in $p(\boldsymbol{x})$, since this could potentially weaken g and thereby compromise safety. Instead a dual projection operator $\forall_x$ is applied which is defined $\forall_x(f) = f'$ if $f' \in Pos$ otherwise $\forall_x(f) = false$ where $f' = f[x \mapsto false] \wedge f[x \mapsto true]$. Note that although $f[x \mapsto false] \vee f[x \mapsto true] \in Pos$ for all $f \in Pos$ it does not follow that $f[x \mapsto false] \wedge f[x \mapsto true] \in Pos$ for all $f \in Pos$. For example, $(x \leftarrow y)[x \mapsto false] \wedge (x \leftarrow y)[x \mapsto true] = \neg y$. Like $\exists_x(f)$, $\forall_x(f)$ eliminates a variable x from f. The fundamental difference is in the direction of approximation in that $\forall_x(f) \models f \models \exists_x(f)$. Thus if Y are the variables that are not present in $p(\boldsymbol{x})$, then $g' = \forall_Y(g)$ eliminates Y from g where $\forall_{\{y_1 \ldots y_n\}}(g) = \forall_{y_1}(\ldots \forall_{y_n}(g))$, whilst strengthening g. A safe calling mode for this particular clause is then given by g', since if g' holds then g holds also.

D_{k+1} will contain a call pattern $\langle p(\boldsymbol{x}), g'' \rangle$ and, assuming $g' \wedge g'' \neq g''$, this is updated with $\langle p(\boldsymbol{x}), g' \wedge g'' \rangle$. Thus the call patterns become progressively stronger on each iteration. Correctness is preserved because call patterns can be safely approximated from below. The space of call patterns forms a complete lattice which ensures that a gfp exists. In fact, because call patterns are approximated from below, the gfp is the most precise solution, and therefore the desired solution. (This contrasts to the norm in logic program analysis where approximation is from above and a lfp is computed). Moreover, since the space of call patterns is finite, termination is assured. In fact, the scheme will converge onto the gfp since (lower Kleene) iteration commences with the top element $D_0 = \{\langle p(\boldsymbol{x}), true \rangle \mid p \in \Pi\}$.

Greatest fixpoint calculation for quicksort Under this procedure quicksort generates the following D_k sequence:

$$D_0 = \left\{ \begin{array}{r} \langle \mathtt{qs}(x_1, x_2, x_3), \mathit{true} \rangle \\ \langle \mathtt{pt}(x_1, x_2, x_3, x_4), \mathit{true} \rangle \\ \langle =<'(x_1, x_2), \mathit{true} \rangle \\ \langle >'(x_1, x_2), \mathit{true} \rangle \end{array} \right\} \quad D_1 = \left\{ \begin{array}{r} \langle \mathtt{qs}(x_1, x_2, x_3), \mathit{true} \rangle \\ \langle \mathtt{pt}(x_1, x_2, x_3, x_4), \mathit{true} \rangle \\ \langle =<'(x_1, x_2), x_1 \wedge x_2 \rangle \\ \langle >'(x_1, x_2), x_1 \wedge x_2 \rangle \end{array} \right\}$$

$$D_2 = \left\{ \begin{array}{r} \langle \mathtt{qs}(x_1, x_2, x_3), \mathit{true} \rangle \\ \langle \mathtt{pt}(x_1, x_2, x_3, x_4), x_2 \wedge (x_1 \vee (x_3 \wedge x_4)) \rangle \\ \langle =<'(x_1, x_2), x_1 \wedge x_2 \rangle \\ \langle >'(x_1, x_2), x_1 \wedge x_2 \rangle \end{array} \right\}$$

$$D_3 = \left\{ \begin{array}{r} \langle \mathtt{qs}(x_1, x_2, x_3), x_1 \rangle \\ \langle \mathtt{pt}(x_1, x_2, x_3, x_4), x_2 \wedge (x_1 \vee (x_3 \wedge x_4)) \rangle \\ \langle =<'(x_1, x_2), x_1 \wedge x_2 \rangle \\ \langle >'(x_1, x_2), x_1 \wedge x_2 \rangle \end{array} \right\}$$

These calculations are non-trivial so consider how D_2 is obtained from D_1 by applying the second (abstract) clause of pt as listed in Figure 2 – the clause with head `pt(T1, M, T2, H)`. The following e_i and g formulae are generated from the demands d_i and the success patterns f_i:

$$e_3 = \mathit{true}$$

$$\begin{aligned} e_2 &= d_2 \wedge (f_2 \rightarrow e_3) \\ &= \mathit{true} \wedge ((\mathtt{Xs} \wedge \mathtt{L} \wedge \mathtt{H}) \rightarrow \mathit{true}) = \mathit{true} \end{aligned}$$

$$\begin{aligned} e_1 &= d_1 \wedge (f_1 \rightarrow e_2) \\ &= (\mathtt{M} \wedge \mathtt{X}) \wedge ((\mathtt{M} \wedge \mathtt{X}) \rightarrow \mathit{true}) = \mathtt{M} \wedge \mathtt{X} \end{aligned}$$

$$\begin{aligned} g &= d \wedge (f \rightarrow e_1) \\ &= \mathit{true} \wedge (((\mathtt{T1} \leftrightarrow \mathtt{X} \wedge \mathtt{Xs}) \wedge (\mathtt{T2} \leftrightarrow \mathtt{X} \wedge \mathtt{L})) \rightarrow (\mathtt{M} \wedge \mathtt{X})) \end{aligned}$$

To characterise those `pt(T1, M, T2, H)` calls which are safe, it is necessary to compute a function g' on the variables `T1`, `M`, `T2`, `H` which, if satisfied by the mode of a call, ensures that g is satisfied by the mode of the call. Put another way, it is necessary to eliminate the variables `X`, `Xs` and `L` from g (those variables which do not occur in the head `pt(T1, M, T2, H)`) to obtain a *Pos* function g' such that g holds whenever g' holds. This is accomplished by calculating $g' = \forall_{\mathtt{L}} \forall_{\mathtt{Xs}} \forall_{\mathtt{X}}(g)$. First consider the computation of $\forall_{\mathtt{X}}(g)$:

$$\begin{aligned} g[\mathtt{X} \mapsto \mathit{false}] &= ((\mathtt{T1} \leftrightarrow \mathit{false} \wedge \mathtt{Xs}) \wedge (\mathtt{T2} \leftrightarrow \mathit{false} \wedge \mathtt{L})) \rightarrow (\mathtt{M} \wedge \mathit{false}) \\ &= (\neg \mathtt{T1} \wedge \neg \mathtt{T2}) \rightarrow \mathit{false} \\ &= \mathtt{T1} \vee \mathtt{T2} \end{aligned}$$

$$\begin{aligned} g[\mathtt{X} \mapsto \mathit{true}] &= ((\mathtt{T1} \leftrightarrow \mathit{true} \wedge \mathtt{Xs}) \wedge (\mathtt{T2} \leftrightarrow \mathit{true} \wedge \mathtt{L})) \rightarrow (\mathtt{M} \wedge \mathit{true}) \\ &= ((\mathtt{T1} \leftrightarrow \mathtt{Xs}) \wedge (\mathtt{T2} \leftrightarrow \mathtt{L})) \rightarrow \mathtt{M} \end{aligned}$$

Since $g[\mathtt{X} \mapsto \mathit{false}] \wedge g[\mathtt{X} \mapsto \mathit{true}] \in \mathit{Pos}$ it follows that:

$$\forall_{\mathtt{X}}(g) = (((\mathtt{T1} \leftrightarrow \mathtt{Xs}) \wedge (\mathtt{T2} \leftrightarrow \mathtt{L})) \rightarrow \mathtt{M}) \wedge (\mathtt{T1} \vee \mathtt{T2})$$

(otherwise $\forall_{\mathtt{X}}(g)$ would be set to false). Eliminating the other variables in a similar way we obtain:

$$\begin{aligned}
\forall_{\mathtt{X}}(g) &= (((\mathtt{T1} \leftrightarrow \mathtt{Xs}) \wedge (\mathtt{T2} \leftrightarrow \mathtt{L})) \rightarrow \mathtt{M}) \wedge (\mathtt{T1} \vee \mathtt{T2}) \\
\forall_{\mathtt{Xs}}\forall_{\mathtt{X}}(g) &= ((\mathtt{T2} \leftrightarrow \mathtt{L}) \rightarrow \mathtt{M}) \wedge (\mathtt{T1} \vee \mathtt{T2}) \\
g' = \forall_{\mathtt{L}}\forall_{\mathtt{Xs}}\forall_{\mathtt{X}}(g) &= \mathtt{M} \wedge (\mathtt{T1} \vee \mathtt{T2})
\end{aligned}$$

Observe that if $\forall_{\mathtt{L}}\forall_{\mathtt{Xs}}\forall_{\mathtt{X}}(g)$ holds then g holds. Thus if the mode of a call satisfies g' then the mode also satisfies g as required. This clause thus yields the call pattern $\langle \mathtt{pt}(x_1, x_2, x_3, x_4), x_2 \wedge (x_1 \vee x_3)\rangle$. Similarly the first and third clauses contribute the patterns $\langle \mathtt{pt}(x_1, x_2, x_3, x_4), \mathit{true}\rangle$ and $\langle \mathtt{pt}(x_1, x_2, x_3, x_4), x_2 \wedge (x_1 \vee x_4)\rangle$. Observe also that

$$\mathit{true} \wedge (x_2 \wedge (x_1 \vee x_3)) \wedge (x_2 \wedge (x_1 \vee x_4)) = x_2 \wedge (x_1 \vee (x_3 \wedge x_4))$$

which gives the final call pattern formula for $\mathtt{pt}(x_1, x_2, x_3, x_4)$ in D_2. The gfp is reached at D_3 since $D_4 = D_3$. The gfp often expresses elaborate calling modes, for example, it states that $\mathtt{pt}(x_1, x_2, x_3, x_4)$ cannot generate an instantiation error (nor any predicate that it calls) if it is called with its second, third and fourth argument ground. This is a surprising result which suggests that the analysis can infer information that might be normally missed by a programmer.

2.4 Work related to mode inference

Mode inference was partly motivated by the revival of interest in logic programming with assertions [4,56,64]. Interestingly, [56] observe that predicates are normally written with an expectation on the initial calling pattern, and hence provide an `entry` assertion to make the, moding say, of the top-level queries explicit. Mode inference gives a way of automatically synthesising `entry` assertions providing a provably correct way of ensuring that instantiation errors do not occur during program execution.

An analysis for type inference could be constructed by refining the analysis presented in this section by replacing the mode domain Pos with the domain of directional types [1,30,40]. This domain is condensing and therefore the domain comes equipped with the relative pseudo-complement operator that is necessary for backward reasoning. Interestingly, type inference can be performed even when the domain is not relatively pseudo-complemented [44]. This, however, relies on a radically different form of fixpoint calculation and therefore this approach to type inference is discussed separately in section 5.

```
split([], [], []) :- true.
split([X | Xs], [X | L1], L2) :- split(Xs, L2, L1).
```

Fig. 4. `split` program expressed in Prolog

3 Backwards termination inference

The aim of termination inference is to determine conditions under which calls to a predicate are guaranteed to terminate [24]. Termination inference is not a new idea in itself; it dates back to the pioneering work of Mesnard and his colleagues [34,50,52,53]. Recently it has been observed, however, that termination inference [24] can be performed by composing backwards analysis with a standard termination checker [10]. The elegance of this approach is that termination analysis can be reversed without dismantling an existing (forwards) termination analysis.

The key advantage of (backwards) termination inference over (forwards) termination checking is that termination inference can deduce, in a single application, a class of queries that lead to finite LD-derivations [24]. To illustrate this key idea, consider the program `split` listed in Figure 4. The `split` predicate arises in the classic mergesort algorithm where it is used to partition a list into sub-lists as preparation for an ordered merge [43]. For instance, the goal `split([a,b,c], L1, L2)` will terminate, binding `L1` and `L2` to `[a,c]` and `[b]` respectively. Correspondingly, a termination checker will ascertain that the call `split(L, L1, L2)` will terminate if `L` is bound to a list of fixed length [43]. However, a termination inference engine such as TerminWeb [24] or cTI [50] will deduce that `split(L, L1, L2)` terminates with either the first argument bound to a closed list or both the second and third arguments bound to closed lists. Of course, a termination checker could be reapplied to prove that `split(L, L1, L2)` will terminate under the latter condition, but inference finds all the termination conditions in one application. Thus termination inference can discover termination conditions that are not observed by one, or possibly many, applications of a termination checker. Note that is not due to a failing of the checker; it is due to the programmer failing to realise that a condition warrants checking. (Actually, the conditions under which termination inference truly generalises termination checking are technical [24] and relate, among other things, to properties of the projection operators $\exists_x$ and $\forall_x$ [40].)

Termination inference can be realised in terms of the backwards analysis framework of [39] that was applied, in the previous section, to the problem of mode inference. In fact the only conceptual difference between mode inference and termination inference is in the way in which assertions are calculated. Whilst for mode analysis assertions are direct groundness abstractions of the builtins, for termination inference, assertions need to be calculated by an analysis of the loops within the program.

Termination analyses typically amount to showing that successive goals in an LD-derivation are decreasing with respect to some well-founded ordering. In the context of a termination checker founded on a binary clause semantics [20], this reduces to observing a size decrease between the arguments of the head and the body atom for each recursive binary clause [10]. From such a checker, a termination inference engine is obtained as follows:

- Firstly, the program is abstracted with respect to a chosen norm (or possibly a series of norms [25]). A norm maps each Herbrand term in the program to a linear expression that represents its size. Syntactic equations between Herbrand terms are replaced with linear inequations which express size relationships. The resulting program is abstract – it is a constraint program over the domain of linear constraints – but it is not binary; abstract clauses may contain more than one body atom.
- Secondly, an abstract version of the binary clause semantics is applied [10]. The (concrete) binary clause semantics of [20] provides a sound basis for termination analysis since the set of (concrete) binary clauses it defines precisely characterises the looping behaviour of the program. Specifically, a call to a given predicate will left-terminate if each corresponding recursive binary clause possesses a body atom that is strictly smaller than its head. Since this set of clauses is not finitely computable, an abstract version of binary clause semantics is used to compute a set of abstract binary clauses which, though finite, faithfully describes the set of concrete binary clauses. The linear inequalities in these abstract clauses capture size relationships between the arguments in the head and the body atom of the concrete clauses.
- Thirdly, combinations of ground arguments that are sufficient for termination are derived. The crucial point is that a decrease in size is only observable if sufficient arguments of a call are ground. These ground argument combinations are extracted from the linear inequalities in the abstract binary clauses, expressed as Boolean functions, and added to the original program in the form of assertions.
- Fourthly and finally, backwards mode analysis is performed on the program augmented with its assertions. The greatest fixpoint then yields groundness conditions which, if satisfied by an initial call, ensure that the call leads to a finite LD-derivation.

Note that backwards termination inference can be considered to be the composition of one black-box that infers binary clauses, with another which extracts assertions from the binary clauses with yet another performs mode inference. Because of this construction, readers who wish to skip the details on approximating loops can progress directly onto section 3.3.

```
subset([], _) :-
   true.
subset([X | Xs], Ys) :-
   member(X, Ys), subset(Xs, Ys).

member(X, [X | Xs]) :-
   true.
member(X, [_ | Ys]) :-
   member(X, Ys).
```

```
subset(0, Ys) :-
   0 ≤ Ys, true.
subset(1 + Xs, Ys) :-
   0 ≤ X, 0 ≤ Xs, 0 ≤ Ys,
   member(X, Ys), subset(Xs, Ys).

member(X, 1 + Xs) :-
   0 ≤ X, 0 ≤ Xs, true.
member(X, 1 + Ys) :-
   0 ≤ X, 0 ≤ Ys, member(X, Ys).
```

Fig. 5. `subset` program expressed in Prolog, and its list-length abstraction

3.1 Program abstraction

Termination inference will be illustrated using the `subset` program listed in the first column of Figure 5. The predicate `subset(L1, L2)` holds iff each element of the list `L1` occurs within the list `L2`. Observe that neither `subset(L1, [a,b,c])` nor the call `subset([a,b,c], L2)` terminate when `L1` and `L2` are uninstantiated. However, both calls will terminate (albeit possibly in failure) when `L1` and `L2` are ground. The challenge is to automatically derive these grounding properties which are sufficient to guarantee termination.

Non-termination of logic programs is the result of infinite loops occurring during execution. Consequently recursive calls are the focus of termination analysis; a logic program will terminate if the arguments of successive calls to a predicate become progressively smaller with respect to a well-founded ordering. Thus, the notion of argument size (and more generally term size) is at the core of termination analyses. To measure term size, a norm is applied which maps a ground term to a natural number. To support program abstraction [28], the concept is normally lifted to terms that contain variables by defining a symbolic norm which maps a term to an expression over variables, non-negative integer constants and the functor $+$. For instance, the list-length norm is defined over the set of ground terms by:

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

whereas the symbolic list-length norm is given by:

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

This symbolic norm describes the length of a list, using a variable to describe the variable length of an open list. For example $|[\mathrm{X} \mid \mathrm{Xs}]|_{\text{length}} = 1 + |\mathrm{Xs}|_{\text{length}}$

= 1 + Xs. Non-list terms are ascribed a length of zero. The second column of Figure 5 gives the list-length norm abstraction of `subset` and `member` in which terms are replaced by their sizes. The abstraction is obtained by replacing each term with its size. Since a norm can only map a variable to a non-negative value, extra inequalities are introduced to ensure that all (size) variables are non-negative. Observe that the resulting abstraction is a constraint program over the system of linear inequations.

3.2 Least fixpoint calculation over binary clauses

In [10] it is shown (using a semantics for call patterns similar to that of [20]) that a logic program is terminating iff its binary unfolding is. Informally, the binary unfolding of a program is the least set of binary clauses each with a head and body such that the head occurs as a head in the original program and, when the original program is called with the head as a goal, then the body occurs as a subsequent sub-goal in an LD-derivation. The binary unfolding is formally expressed in terms of the lfp of a T_P-style operator [10,20]. Moreover, an abstract binary unfolding can be obtained by applying an abstraction of the binary unfolding operator to the abstract program [10].

Calculation of this abstract lfp is complicated by the property that the domain of linear inequations does not satisfy the ascending chain condition [15]. This property compromises the termination of the abstract lfp calculation since it enables a set of abstract binary clauses to be repeatedly enlarged on successive iterates *ad infinitum*. Termination can be assured, however, by restricting the inequations that occur within abstract binary clauses to a finite sub-class, for example, the sub-class of monotonicity and equality constraints [5]. Alternatively widening can be applied to enforce convergence [3]. Using the latter technique the following set of abstract binary unfolding is obtained:

$$\left\{\begin{array}{l}
\texttt{member}(x_1, x_2) \text{ :- } 0 \leq x_1 \wedge 1 \leq x_2, \texttt{true} \\
\texttt{member}(x_1, x_2) \text{ :- } 0 \leq x_1 \wedge 0 \leq y_2 \wedge 1 + y_2 \leq x_2 \wedge x_1 = y_1, \\
\qquad\qquad\qquad\quad \texttt{member}(y_1, y_2) \\
\texttt{subset}(x_1, x_2) \text{ :- } 0 \leq x_1 \wedge 0 \leq x_2, \texttt{true} \\
\texttt{subset}(x_1, x_2) \text{ :- } y_2 \leq x_2 \wedge 1 \leq x_1 \wedge 0 \leq y_2 \wedge 0 \leq y_1, \\
\qquad\qquad\qquad\quad \texttt{member}(y_1, y_2) \\
\texttt{subset}(x_1, x_2) \text{ :- } y_1 + 1 \leq x_1 \wedge y_2 = x_2 \wedge 1 \leq x_1 \wedge 1 \leq x_2, \\
\qquad\qquad\qquad\quad \texttt{subset}(y_1, y_2)
\end{array}\right\}$$

The set of abstract clauses contains at most $|\Pi|^2$ clauses where Π is the set of predicate symbols occurring in the program (which is assumed to include `true`). This follows because widening ensures that two abstract clauses cannot share the same predicate symbols in both the head and body. Note that if an abstract binary clause has `true` as its body atom then the clause does not describe a loop and therefore has no bearing on the termination behaviour. Such clauses are given above simply for completeness.

```
subset(A, B) :-                 subset(A, B) :-
   tell(A = []),                   ask(A), tell(A),
   true.                           true.
subset(A, B) :-                 subset(A, B) :-
   tell(A = [X | Xs]),             ask(A), tell(A ↔ (X ∧ Xs)),
   member(X, B),                   member(X, B),
   subset(Xs, B).                  subset(Xs, B).

member(X, B) :-                 member(A, B) :-
   tell(B = [X | Xs]),             ask(B), tell(B ↔ (X ∧ Xs)),
   true.                           true.
member(A, B) :-                 member(A, B) :-
   tell(B = [Y | Ys]),             ask(B), tell(B ↔ (Y ∧ Ys)),
   member(A, Ys).                  member(A, Ys).
```

Fig. 6. `subset` normalised and as a *Pos* abstraction with assertions

3.3 Extracting the assertions from the binary clauses

Those abstract binary clauses that involve recursive calls are as follows:

$$\begin{array}{ll}
\texttt{member}(x_1, x_2) \text{ :-} & \texttt{subset}(x_1, x_2) \text{ :-} \\
\quad 0 \le x_1 \ \wedge\ 0 \le y_2 \ \wedge & \quad y_1 + 1 \le x_1 \wedge y_2 = x_2 \ \wedge \\
\quad 1 + y_2 \le x_2 \ \wedge\ x_1 = y_1, & \quad 1 \le x_1 \wedge 1 \le x_2, \\
\quad \texttt{member}(y_1, y_2). & \quad \texttt{subset}(y_1, y_2).
\end{array}$$

Consider the abstract clause for member. The inequality $1 + y_2 \le x_2$ asserts that the recursive call is smaller than the previous call (as measured by the list-length norm). Therefore, assuming that the second argument of the original call to member is ground, each recursive call will operate on a strictly smaller list and thus terminate. Hence, although one abstract member clause approximates many concrete member clauses, the approximation is sufficiently precise to enable termination properties to be deduced. Likewise, the inequality $y_1 + 1 \le x_1$ for `subset` ensures that termination follows if the first argument of the initial call to `subset` is ground.

Since termination is dependent on groundness, the inequalities in recursive abstract clauses induce groundness requirements that, if satisfied, assure termination. Since the number of ground argument combinations is exponential in the number of arguments, inferring the optimal set of ground argument combinations is potentially expensive (though experimentation suggests the contrary [51]). Therefore a subset of the argument combinations may only be considered [24]. Once extracted, the requirements are added to the original logic program in the form of assertions. Figure 6 lists the `subset` program complete with assertions that are sufficient for termination.

3.4 Backwards mode analysis

Backwards mode analysis can then be performed on the program with its assertions as specified in the previous section. Using the notation from that section, backwards mode analysis yields the following sequence of iterates:

$$D_0 = \left\{ \begin{matrix} \langle \texttt{member}(x_1, x_2), \mathit{true} \rangle \\ \langle \texttt{subset}(x_1, x_2), \mathit{true} \rangle \end{matrix} \right\} \qquad D_1 = \left\{ \begin{matrix} \langle \texttt{member}(x_1, x_2), x_2 \rangle \\ \langle \texttt{subset}(x_1, x_2), x_1 \rangle \end{matrix} \right\}$$

$$D_2 = \left\{ \begin{matrix} \langle \texttt{member}(x_1, x_2), x_2 \rangle \\ \langle \texttt{subset}(x_1, x_2), x_1 \wedge x_2 \rangle \end{matrix} \right\}$$

The fixpoint is reached and checked in the next iteration since $D_3 = D_2$. The fixpoint specifies grounding conditions that are sufficient for termination. That is, `subset` is guaranteed to left-terminate if both of its arguments are ground. This is as expected, since the first argument of `subset` needs to be ground in order that its own recursive call terminates. Moreover, the second argument additionally needs to be ground in order that the call to `member` terminates. Both the recursive call and the call to `member` are required to terminate to assure that a call to the second clause of `subset` terminates.

3.5 Work related to termination inference

Performing termination inference via backwards analysis is a comparatively new idea [24] but termination inference was developed by Mesnard and others [34,50–53] long before this connection was made. Their system, the cTI analyser, applied a μ-calculus solver to compute a greatest fixpoint. This seems to suggest that greatest fixpoints are intrinsic to the problem itself. On the other hand, the termination inference analyser reported in [24] (and described in this section) is composed from two components: a standard termination checker [10] and a backwards analysis. The resulting analyser is similar to cTI; the main difference is its design as two existing black-box components which, according to [24], simplifies the formal justification and implementation.

4 Backwards suspension inference

In mode inference [39] the assertions are synthesised from the builtins. In termination inference [24] the assertions are distilled from a separate analysis of the loops which occur within the program [10]. Both these analyses share the same backwards analysis component – a component which essentially propagates requirements right-to-left over sequences of goals against the control-flow. Interestingly, and perhaps surprisingly, backwards analysis can still be applied when the control is more loosely defined. In fact, backwards analysis is still applicable even when the control is specified by a delay mechanism which blocks the selection of a sub-goal until some condition is satisfied [7]. This, arguably, is one of the most flexible ways of specifying control within logic programming.

Delays have proved to be invaluable for handling negation [54], delaying non-linear constraints [32], enforcing termination [47], improving search and modelling concurrency [45]. However, reasoning about logic programs with delays is notoriously difficult and one reoccurring problem for the programmer is that of determining whether a given program and goal can reduce to a state which possesses a sub-goal that suspends indefinitely. A number of abstract interpretation schemes [8,11,17] have therefore been proposed for verifying that a program and goal cannot suspend in this fashion. These analyses are essentially forwards in that they simulate the operational semantics tracing the execution of the program in the direction of the control with collections of abstract states. This section reviews a suspension analysis that is performed backwards by propagating requirements against the control-flow. Specifically, rather than verifying that a particular goal will not lead to a suspension, the analysis infers a class of goals that will not lead to suspension. This approach has the computational advantage that the programmer need not rerun the analysis for different (abstract) queries. Moreover, like the previous analyses, this suspension analysis is formulated as two simple bottom-up fixpoint computations. The analysis strikes a good balance between tractability and precision. It avoids the complexity of goal interleaving by exploiting reordering properties of monotonic and positive Boolean functions.

Another noteworthy aspect of the analysis is that it verifies whether a logic program with delays can be scheduled with a *local* selection rule [63]. Under local selection, the selected atom is completely resolved, that is, those atoms it directly and indirectly introduces are also resolved, before any other atom is selected. Leftmost selection is one example of local selection. Knowledge about suspension within the context of local selection is useful within it own right [17,41]. In particular, [17] explains how various low-level optimisations, such as returning output values in registers, can be applied if goals can be scheduled left-to-right without suspension. Furthermore, any program that can be shown to be suspension-free under local selection is clearly suspension-free with a more general selection rule. Note, however, that the converse does not follow and the analysis cannot infer non-suspension if the program relies on coroutining techniques.

4.1 Worked example on suspension inference

To illustrate the ideas behind suspension analysis, consider an analysis of the Prolog program listed in Figure 7. 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(-, ?, -)`

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

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

Fig. 7. `inorder` program in expressed in Prolog with block declarations

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 problem is to compute input modes which are sufficient to guarantee that any `inorder` query which satisfies the modes will not lead to a suspension under local selection. This problem can be solved with backwards analysis. Backwards analysis infers requirements on the input which ensure that certain properties hold at (later) program points [39]. Exactly like before, the analysis is tackled via an abstraction step followed by a least fixpoint (lfp) and then a greatest fixpoint (gfp) computation.

4.2 Program abstraction

Abstraction reduces to two transformations: one from a Prolog with delay program to a concurrent constraint programming [60] (ccp) program and another from the ccp program to a *Pos* abstraction. The Prolog program is re-written to a ccp program to make blocking requirements explicit in the program as ask constraints. More exactly, a clause of a ccp program takes the form h :- $\mathsf{ask}(c')$, $\mathsf{tell}(c'')$, g where h is an atom, g is a conjunction of body atoms and c' and c'' are the ask and tell constraints. The asks are guards that inspect the store and specify synchronisation behaviour whereas the tells are writes that update the store. As before, empty conjunctions of atoms are denoted by true. Unlike before, ask does not denote an assertion but a synchronisation requirement. Moreover, a conjunction of goals g is not necessarily executed left-to-right: goals can only be reduced with a clause when the ask constraint within the clause is satisfied. A goal will suspend until this is the case, hence the execution order of the sub-goals within a goal does not necessarily concur with the textual (left-to-right) ordering of these sub-goals. In this particular example, the only ask constraint that appears in the program is $\mathtt{nonvar}(x)$ which formalises the requirement that x must be bound to a non-variable term.

The second transform abstracts the ask and tell constraints with Boolean functions which capture instantiation dependencies. The ask constraints are

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

`append(L, Ys, A) :-`
`   ask(nonvar(L) ∨ nonvar(A)),`
`   tell(L = [], A = Ys),`
`   true.`
`append(L, Ys, A) :-`
`   ask(nonvar(L) ∨ nonvar(A)),`
`   tell(L = [X|Xs], A = [X|Zs]),`
`   append(Xs, Ys, Zs).`
	`append(L, Ys, A) :-`
	`   ask(L ∨ A),`
	`   tell(L ∧ (A ↔ Ys)),`
	`   true.`
	`append(L, Ys, A) :-`
	`   ask(L ∨ A),`
	`   tell(L ↔ (X ∧ Xs),A ↔ (X ∧ Zs)),`
	`   append(Xs, Ys, Zs).`

Fig. 8. `inorder` program expressed in ccp and as a *Pos* abstraction

abstracted from below whereas the tell constraints are abstracted from above. More exactly, an ask abstraction is stronger than the ask constraint – whenever the abstraction holds then the ask constraint is satisfied; whereas the tell abstraction is weaker than the tell constraint – whenever the tell constraint holds then so does its abstraction. For example, the function $\texttt{L} \vee \texttt{A}$ describes states where either `L` or `A` is ground [2] which, in turn, ensure that the ask constraint `nonvar(L)` $\vee$ `nonvar(A)` holds. On the other hand, once the tell $\texttt{A} = [\texttt{V}|\texttt{RI}]$ holds, then the grounding behaviour of the state (and all subsequent states) is described by $\texttt{A} \leftrightarrow (\texttt{V} \wedge \texttt{RI})$.

4.3 Least fixpoint calculation

The least fixpoint calculation approximates the success patterns of the ccp program (and thus the Prolog with delays program) by mimicking the T_P operator [28]. A success pattern is an atom with distinct variables for arguments paired with a *Pos* formula over those variables. This is the same notion of success pattern as used in mode inference and, just as in mode inference, a success pattern summarises the behaviour of an atom by describing the bindings it can make. The lfp of the *Pos* program can be computed in a finite number of iterates to give the following lfp:

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

4.4 Greatest fixpoint calculation

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

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

and incrementally strengthens the call pattern formulae until they are safe, that is, they describe queries which are guaranteed not to violate the ask constraints. The iterate D_{i+1} is computed by putting $D_{i+1} = D_i$ and then revising D_{i+1} by considering each $p(\boldsymbol{x})$:- $\mathsf{ask}(d), \mathsf{tell}(f), p_1(\boldsymbol{x}_1), \ldots, p_n(\boldsymbol{x}_n)$ in the abstract program and calculating a (monotonic) formula that describes input modes (if any) under which the atoms in the clause can be scheduled without suspension under local selection. A monotonic formula over set of variables X is any formula of the form $\vee_{i=1}^{n}(\wedge Y_i)$ where $Y_i \subseteq X$ [18]. Let d_i denote a monotonic formula that describes the call pattern requirement for $p_i(\boldsymbol{x}_i)$ in D_i and let f_i denote the success pattern formula for $p_i(\boldsymbol{x}_i)$ in the lfp (that is not necessarily monotonic). A new call pattern for $p(\boldsymbol{x})$ is computed using the following algorithm:

- Calculate $e = \wedge_{i=1}^{n}(d_i \rightarrow f_i)$ that describes the grounding behaviour of the compound goal $p_1(\boldsymbol{x}_1), \ldots, p_n(\boldsymbol{x}_n)$. The intuition is that $p_i(\boldsymbol{x}_i)$ can be described by $d_i \rightarrow f_i$ since if the input requirements d_i hold then $p_i(\boldsymbol{x}_i)$ can be executed without suspension, hence the output f_i must also hold.
- Compute $e' = \wedge_{i=1}^{n} d_i$ which describes a groundness property sufficient for scheduling all of the goals in the compound goal without suspension. Then $e \rightarrow e'$ describes a grounding property which, if satisfied, when the compound goal is called ensures the goal can be scheduled by local selection without suspension.
- Calculate $g = d \wedge (f \rightarrow (e \rightarrow e'))$ that describes a grounding property which is strong enough to ensure that both the ask is satisfied and the body atoms can be scheduled by local selection without suspension.
- Eliminate those variables not present in $p(\boldsymbol{x})$, Y say, by calculating $g' = \forall_Y(g)$ where $\forall_{\{y_1 \cdots y_n\}}(g) = \forall_{y_1}(\ldots \forall_{y_n}(g))$. Hence $\forall_x(f)$ entails f and g' entails g, so that a safe calling mode for this particular clause is then given by g'.
- Compute a monotonic function g'' that entails g'. Since g'' is stronger than g' it follows that g'' is sufficient for scheduling the compound goal by local selection without suspension. The function g' needs to be approximated by a monotonic function since the $e \rightarrow e'$ step relies on d_i being monotonic.
- Replace the pattern $\langle p(\boldsymbol{x}), g''' \rangle$ in D_{i+1} with $\langle p(\boldsymbol{x}), g'' \wedge g''' \rangle$.

This procedure generates the following D_i sequence:

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

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

The gfp is reached and checked in three iterations. The result asserts that a local selection rule exists for which `inorder` will not suspend if either its first or second arguments are ground. Indeed, observe that if the first argument is ground then body atoms of the second `inorder` clause can be scheduled as follows: `inorder(L, LI)`, then `inorder(R, RI)`, and then `append(LI, A, I)`. Conversely, if the second argument is ground, then the reverse ordering is sufficient for non-suspension. These call patterns are intuitive and experimental evaluation [26] suggests that unexpected and counter-intuitive call patterns arise (almost exclusively) in buggy programs. This suggests that the analysis has a useful rôle in bug detection and program development.

4.5 Work related to suspension inference

One of the most closely related works comes surprisingly from the compiling control literature and in particular the problem of *generating* a local selection rule under which a program universally terminates [34]. The technique of [34] builds on the termination inference method of [50] which infers initial modes for a query that, if satisfied, ensure that a logic program left-terminates. The chief advance in [34] over [50] 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 $n(n-1)$ 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 [50] is then adapted to include consistency constraints among the $b_{i,j}$, for instance, $b_{j,k} \wedge \neg b_{i,k} \rightarrow \neg b_{i,j}$. In addition, the $b_{i,j}$ are used to determine whether the post-conditions of a_i contribute to the pre-conditions of a_j. Although motivated differently and realised differently (in terms of the Boolean μ-calculus) this work also uses Boolean functions to finesse the problem of enumerating the goal reorderings.

A demand analysis for the ccp language Janus [61] is proposed in [16] which determines whether or not a predicate is uni-modal. A predicate is uni-modal iff the argument tuple for each clause shares the same minimal pattern of instantiation necessary for reduction. The demand analysis of a predicate simply traverses the head and guard of each clause to determine the extent to which arguments have to be instantiated. Body atoms need not be considered so the analysis does not involve a fixpoint computation. A related paper [17] presents a goal-dependent (forwards) analysis that detects those ccp predicates which can be scheduled left-to-right without deadlock. This work 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 surprisingly effective because of the way data often flows left-to-right.

5 Backwards type inference

Backwards mode inference, termination inference and suspension inference analysis of the previous sections all apply the same operator to model reversed information flow – the relative pseudo-complement. The key idea that these analyses exploit is that if d_2 expresses a set of requirements that must hold after a constraint is added to the store, and d_1 models the constraint itself, then $d_1 \rightarrow d_2$ expresses the requirements that must hold on the store before the constraint. Comparatively few domains possess a relative pseudo-complement and, arguably, the most well-known type domain that comes equipped with a relative pseudo-complement operator is the domain of directional types [1,30]. This section demonstrates that backwards analysis is still applicable to problems in program development even when the domain is not relatively pseudo-complemented or when the relative pseudo-complement is not particularly tractable [40]. The section focuses on the problem of inferring type signatures for predicates that are sufficient to ensure that the execution of the program with a query satisfying the inferred type signatures will be free from type errors. This problem generalises backwards mode inference – types are richer than modes. It also generalises type checking in which the programmer declares type signatures for all predicates in the program and a type checker verifies that the program is well-typed with respect to these type signatures, that is, these type signatures are consistent with the operational semantics of the program.

The value of type inference is illustrated by returning to the quicksort program listed in Figure 1. A type checker would require the programmer to declare type signatures for `qs` and `pt` and then check if the program is well-typed with respect to these and the type signatures for builtin predicates $\leq$ and $>$ stipulated in the user manual. In contrast, type signature inference will infer that if `qs` is called with a list of numbers as the first argument then the execution of the program will not violate the type signatures of $\leq$ and $>$; the programmer need not declare types for `qs` nor `pt`. Backwards type analysis gives the programmer the flexibility not to declare and maintain type signatures for predicates that are subject to frequent modifications during program development. In the extreme situation, the programmer may choose to leave unspecified type signatures for all user-defined predicates and let the analyser to infer type signatures from builtin and library predicates. One application of the new analysis is automatic program documentation. Type signatures provide valuable information for both program development and maintenance [62]. Another application is in bug detection. The inferred type signature for a predicate can be compared with that intended by the programmer and any discrepancy indicates the possible existence of bugs.

5.1 Greatest fixpoint calculation

The analysis is performed by computing a greatest fixpoint. It starts by assuming that no call causes a type error and then checks this assumption by reasoning backwards over all clauses. If an assertion is violated, pre-conditions are strengthened, and the whole process is repeated. The basic datum of the analysis is a type constraint. A type constraint is a disjunction of conjunctive type constraints. A conjunctive type constraint, in turn, is a conjunction of atomic type constraints of the form $x{:}\tau$ where x is a variable and τ a type that denotes a set of terms closed under instantiation. Similarly to before, tell constraints distinguish syntactic equations from assertions which are themselves indicated by ask constraints. The assertions specify type constraints which must be respected by the execution of the program.

Each clause of the normalised program takes the form of $p(\boldsymbol{x})$:- $B_1, \ldots, B_k$ where each B_i is either:

- an assertion $\mathsf{ask}(\phi)$ where ϕ is a type constraint or
- $\mathsf{tell}(E)$ where E is a syntactic equation (unification) or
- a call to an atom $q(\boldsymbol{y})$ where $\boldsymbol{y}$ is a vector of distinct variables.

Unlike previously, ask and tell constraints can occur multiply within the same clause. A conjunction of body atoms $B_1, \ldots, B_k$ is executed left-to-right.

As previously, backwards analysis reduces to computing a finite sequence of iterates D_i. Each D_i is a mapping from an atom $p(\boldsymbol{x})$ to a function that itself maps a type constraint ϕ_R to another ϕ_L such that the execution of $p(\boldsymbol{x})$ in a state satisfying ϕ_L succeeds (if it does) only in a state satisfying ϕ_R and respects type constraints given by the assertions. The pair $\langle p(\boldsymbol{x}), \phi_R\rangle$ is called a demand whereas ϕ_L is a pre-condition for $\langle p(\boldsymbol{x}), \phi_R\rangle$. D_{i+1} is computed from D_i by updating the pre-condition for each demand in D_i and adding new demands to D_{i+1} if necessary. For a demand $\langle p(\boldsymbol{x}), \phi_R\rangle$, a type constraint ϕ_L^C is computed from each clause $p(\boldsymbol{x})$:- $B_1, \ldots, B_k$ by computing a series $\psi_k, \ldots, \psi_0$ of type constraints. This starts by assigning $\psi_k = \phi_R$. Then every other ψ_{j-1} is computed from ψ_j as follows:

- If $B_j = \mathsf{ask}(\phi)$ then ψ_{j-1} is calculated by $\psi_{j-1} = \psi_j \wedge \phi$.
- If $B_j = \mathsf{tell}(E)$ then ψ_{j-1} is computed by performing backwards abstract unification. Backwards abstract unification ensures that the result of unifying E in the context of a store satisfying ψ_{j-1} is a store satisfying ψ_j. Since the domain is not condensing, backwards unification cannot coincide with the relative pseudo-complement operator. The relative pseudo-complement operator is unique in that it delivers the weakest abstraction which when combined with one given abstraction, entails another given abstraction. This suggests there may exist a pre-condition which is strictly weaker than ψ_{j-1} or strictly incomparable with ψ_{j-1} which is also sufficient for ensuring that ψ_j holds after E. Put another way, backwards unification does not come with the precision guarantee that characterises the relative pseudo-complement.

- If $B_j = q(\boldsymbol{y})$ is a call to a user-defined predicate then ψ_{j-1} is computed as follows:
  - Let $\psi_j = \bigvee_{l=1}^{m} \mu_l$ where each μ_l is a conjunctive type constraint.
  - Apply existential quantification to project μ_l onto the variables $\boldsymbol{y}$ to obtain ν_l. Hence ν_l is weaker than μ_l, that is, ν_l holds if μ_l holds. Moreover, $\langle q(\boldsymbol{y}), \nu_l \rangle$ is a demand that constrains only variables in $\boldsymbol{y}$.
  - If $\langle q(\boldsymbol{y}), \nu_l \rangle$ (modulo renaming) is in D_i then $\omega_l = D_i(\langle q(\boldsymbol{y}), \nu_l \rangle)$; recall that D_i is interpreted as a mapping from demands to pre-conditions.
  - Otherwise, $\omega_l = \mathit{true}$ and $\langle q(\boldsymbol{y}), \nu_l \rangle \mapsto \mathit{true}$ is added into D_{i+1}, thereby introducing a new demand.
  - Put $\psi_{j-1} = \bigvee_{l=1}^{m} (\omega_l \wedge \upsilon_l)$ where each υ_l is obtained from μ_l by applying existential quantification to project out variables in $\boldsymbol{y}$.

  Then ψ_{j-1} is a pre-condition for $\langle q(\boldsymbol{y}), \psi_j \rangle$ provided that ω_l is a pre-condition for $\langle q(\boldsymbol{y}), \nu_l \rangle$ for each l.

Finally ϕ_L^C is computed from ψ_0 via universal quantification by projecting onto the variables within $p(\boldsymbol{x})$. As in the previous backwards analyses, this strengthens the pre-condition such that ψ_0 holds if ϕ_L^C holds.

5.2 Worked example on type inference

To illustrate, consider the `insertionsort` program listed in the first column of Figure 9. The second column gives the program in a normalised form, decorated with ask and tell constraints. Note how the tests $\mathtt{X} > \mathtt{Y}$ and $\mathtt{X} \leq \mathtt{Y}$ are both replaced with the tell constraint $\mathtt{X}\text{:num} \wedge \mathtt{Y}\text{:num}$ where num denotes the set of numbers. Unlike the previous backwards analyses, the analysis is driven from an initial demand. The initial demand is the pair $\langle \mathtt{sort(Xs, Ys)}, \mathit{true} \rangle$ for which a pre-condition is required, hence D_0 is:

$$D_0 = \{ \langle \mathtt{sort(Xs, Ys)}, \mathit{true} \rangle \mapsto \mathit{true} \}$$

The iterate D_1 is computed by successively updating D_0 by considering each clause in turn. To illustrate, consider the first clause for `sort` where $B_1 = \mathtt{append(As, Cs, Xs)}$, $\ldots$, $B_6 = \mathtt{sort(Zs, Ys)}$. This clause has 6 body atoms and analysis amounts to computing $\psi_5, \ldots, \psi_0$ where $\psi_6 = \mathit{true}$. The analysis proceeds as follows:

- Firstly, $\psi_5 = \mathit{true}$ is computed. Since $B_6 = \mathtt{sort(Zs, Ys)}$ is an atom, ψ_6 is projected onto the variables $\{\mathtt{Zs}, \mathtt{Ys}\}$, yielding true. Then D_1 is checked for the demand $\langle B_6, \mathit{true} \rangle$. Because $\langle \mathtt{sort(Zs, Ys)}, \mathit{true} \rangle$ is a variant of $\langle \mathtt{sort(Xs, Ys)}, \mathit{true} \rangle$, no new demand is added to D_1 and thus $\psi_5 = D_1(\langle \mathtt{sort(Zs, Ys)}, \mathit{true} \rangle) = \mathit{true}$.

```
sort(Xs, Ys) :-
   append(As, [X, Y|Bs], Xs),
   X > Y,
   append(As, [Y, X|Bs], Zs),
   sort(Zs, Ys).
sort(Xs, Xs) :-
   order(Xs).

append([], Ys, Ys) :- true.
append([X|Xs], Ys, [X|Zs]) :-
   append(Xs, Ys, Zs).

order([]) :- true.
order([_]) :- true.
order([X, Y|Xs]) :-
   X ≤ Y,
   order([Y|Xs]).
```

```
sort(Xs, Ys) :-
   append(As, Cs, Xs),
   tell(Cs = [X, Y|Bs]),
   ask(X:num ∧ Y:num),
   tell(Ds = [Y, X|Bs]),
   append(As, Ds, Zs),
   sort(Zs, Ys).
sort(Xs, Ys) :-
   tell(Xs = Ys),
   order(Xs).

append(Xs, Ys, Zs) :-
   tell(Xs = [], Ys = Zs).
append(Xs, Ys, Zs) :-
   tell(Xs = [X|Xs1], Zs = [X|Zs1]),
   append(Xs1, Ys, Zs1).

order(Xs) :-
   tell(Xs = []).
order(Xs) :-
   tell(Xs = [_]).
order(Xs) :-
   tell(Xs = [X|Xs1], Xs1 = [Y|Ys]),
   ask(X:num ∧ Y:num),
   order(Xs1).
```

Fig. 9. `insertionsort` expressed in Prolog and with type assertions

- Secondly, $\psi_4 = true$ is computed. As previously $B_5 = \texttt{append}(\texttt{As}, \texttt{Ds}, \texttt{Zs})$ is an atom. Thus $true$ is projected onto $\{\texttt{As}, \texttt{Ds}, \texttt{Zs}\}$, obtaining $true$. Unlike before, D_1 does not contain the demand $\langle B_5, true\rangle$ and therefore D_1 is updated to

$$D_1 = \left\{ \begin{array}{r} \langle \texttt{append}(\texttt{As}, \texttt{Ds}, \texttt{Zs}), true\rangle \mapsto true, \\ \langle \texttt{sort}(\texttt{Xs}, \texttt{Ys}), true\rangle \mapsto true \end{array} \right\}$$

  and $\psi_4 = D_1(\langle \texttt{append}(\texttt{As}, \texttt{Ds}, \texttt{Zs}), true\rangle) = true$.
- Thirdly, $\psi_3 = true$ is computed. Because $B_4 = \mathsf{tell}(\texttt{Ds} = [\texttt{Y}, \texttt{X}|\texttt{Bs}])$, backwards abstract unification is applied. Since $\psi_4 = true$, this requirement is trivially satisfied, hence $\psi_3 = true$.
- Fourthly, $\psi_2 = \texttt{X}{:}\mathrm{num} \wedge \texttt{Y}{:}\mathrm{num}$ is computed. Since $B_3 = \mathsf{ask}(\texttt{X}{:}\mathrm{num} \wedge \texttt{Y}{:}\mathrm{num})$, $\psi_2 = \psi_3 \wedge \phi$ where $B_3 = \mathsf{ask}(\phi)$.
- Fifthly, $\psi_1 = (\texttt{X}{:}\mathrm{num} \wedge \texttt{Y}{:}\mathrm{num}) \vee \texttt{Cs}{:}\mathrm{list}(\mathrm{num})$ is computed where list is the standard polymorphic list constructor associated with the typing rules $\mathrm{list}(\beta) ::= []$ and $\mathrm{list}(\beta) ::= [\beta|\mathrm{list}(\beta)]$. Abstract backwards unification is

applied since $B_2 = \mathsf{tell}(\mathtt{Cs} = [\mathtt{X},\mathtt{Y}|\mathtt{Bs}])$. The conjunct $(\mathtt{X}\text{:num} \wedge \mathtt{Y}\text{:num})$ derives from the fact that a type constraint that holds before unification also holds after unification. The conjunct $\mathtt{Cs}$:list(num) derives from the fact that both Cs and $[\mathtt{X},\mathtt{Y}|\mathtt{Bs}]$ are of the same type after unification and $\mathtt{Cs}$:list(num) implies $[\mathtt{X},\mathtt{Y}|\mathtt{Bs}]$:list(num), hence $(\mathtt{X}\text{:num} \wedge \mathtt{Y}\text{:num})$. More generally, backwards abstract unification takes as inputs an equational constraint E and a type constraint ψ and produces as output a type constraint ϕ which describes θ whenever ψ describes $mgu(\theta(E)) \circ \theta$.

- Sixthly, $\psi_0 = true$ is computed. Since $B_1 = \mathtt{append(As, Cs, Xs)}$ is an atom, $(\mathtt{X}\text{:num} \wedge \mathtt{Y}\text{:num})$ is projected onto $\{\mathtt{As}, \mathtt{Cs}, \mathtt{Xs}\}$ yielding $true$. A variant of $\langle \mathtt{append(As, Cs, Xs)}, true\rangle$ is contained within D_1. However, projecting $\{\mathtt{As}, \mathtt{Cs}, \mathtt{Xs}\}$ out of $(\mathtt{X}\text{:num} \wedge \mathtt{Y}\text{:num})$ yields $(\mathtt{X}\text{:num} \wedge \mathtt{Y}\text{:num})$. Thus, one pre-condition for $\langle \mathtt{append(As, Cs, Xs)}, (\mathtt{X}\text{:num} \wedge \mathtt{Y}\text{:num})\rangle$ is $(true \wedge (\mathtt{X}\text{:num} \wedge \mathtt{Y}\text{:num})) = (\mathtt{X}\text{:num} \wedge \mathtt{Y}\text{:num})$. Another is obtained by projecting $\mathtt{Cs}$:list(num) onto $\{\mathtt{As}, \mathtt{Cs}, \mathtt{Xs}\}$ to obtain $\mathtt{Cs}$:list(num), hence D_1 is updated with the new demand:

$$D_1 = \left\{ \begin{array}{r} \langle \mathtt{append(As, Ds, Zs)}, true\rangle \mapsto true, \\ \langle \mathtt{append(As, Cs, Xs)}, \mathtt{Cs}\text{:list(num)}\rangle \mapsto true, \\ \langle \mathtt{sort(Xs, Ys)}, true\rangle \mapsto true \end{array} \right\}$$

  Because $D_1(\langle \mathtt{append(As, Cs, Xs)}, \mathtt{Cs}\text{:list(num)}\rangle) = true$, the other pre-condition for $\langle \mathtt{append(As, Cs, Xs)}, \mathtt{Cs}\text{:list(num)}\rangle$ is $true$. Therefore $\psi_0 = (\mathtt{X}\text{:num} \wedge \mathtt{Y}\text{:num}) \vee true = true$.

Processing the second clause of `sort` gives the same pre-condition $true$ and introduces one more demand $\langle \mathtt{order(Xs)}, true\rangle$. Therefore

$$D_1 = \left\{ \begin{array}{r} \langle \mathtt{append(As, Ds, Zs)}, true\rangle \mapsto true, \\ \langle \mathtt{append(As, Cs, Xs)}, \mathtt{Cs}\text{:list(num)}\rangle \mapsto true, \\ \langle \mathtt{order(Xs)}, true\rangle \mapsto true, \\ \langle \mathtt{sort(Xs, Ys)}, true\rangle \mapsto true \end{array} \right\}$$

Omitting details of the remaining computation, the gfp is reached at D_5 with

$$D_5 = \left\{ \begin{array}{l} \qquad\quad \langle \mathtt{append(Xs, Ys, Zs)}, true\rangle \mapsto true, \\ \langle \mathtt{append(Xs, Ys, Zs)}, \mathtt{Zs}\text{:list(num)}\rangle \mapsto \mathtt{Zs}\text{:list(num)}, \\ \langle \mathtt{append(Xs, Ys, Zs)}, \mathtt{Ys}\text{:list(num)}\rangle \mapsto \\ \qquad\qquad\qquad\qquad \mathtt{Ys}\text{:list(num)} \vee \mathtt{Zs}\text{:list(num)}, \\ \qquad\qquad\qquad \langle \mathtt{order(Xs)}, true\rangle \mapsto \mathtt{Xs}\text{:list(num)}, \\ \qquad\qquad \langle \mathtt{sort(Xs, Ys)}, true\rangle \mapsto \mathtt{Xs}\text{:list(num)} \end{array} \right\}$$

The gfp asserts that `sort` cannot generate a type error (nor any predicate it subsequently calls) if it is called with a list of numbers as its first argument. It also states that `order` will not generate a type error if it called with a list of numbers. Interestingly, it also asserts that calling `append` with its third argument instantiated to a list of numbers ensures that its second argument is instantiated to a list of numbers.

5.3 Work related to type inference

Type analysis can be performed either with or without type definitions provided by the programmer. The former are easy for the programmer to understand whereas the latter are useful in compiler optimisation but can be more difficult for the programmer to interpret. If type definitions are not given by the programmer, then the analysis has to infer both the type definitions and the type descriptions for the program components. Traditionally unary regular logic programs [66] and type graphs [13] have been applied to this class of problem, though modern set-based techniques founded on non-deterministic finite tree automata offer a number of advantages [23].

Alternatively, if type definitions are supplied by the programmer, then the analysis need only infer type descriptions from the type constructors for the program components. In this class of problem of particular note is the work on formulating type dependency domains with ACI-unification [9] since the resulting domains condense. Directional type analysis [1,59] is likewise performed with type definitions provided by the programmer. A directional type $p(\boldsymbol{x}) : \sigma \to \tau$ indicates that if $p(\boldsymbol{x})$ is called with $\boldsymbol{x}$ being of type σ then $\boldsymbol{x}$ is of type τ upon the success of $p(\boldsymbol{x})$. Aiken and Lakshman [1] provide a procedure for checking if a program is well-typed with respect to a given set of monomorphic directional types, whereas Rychlikowski and Truderung [59] provide type checking and inference algorithms for polymorphic types.

All the above type analyses propagate type information in the direction of program execution and compute upper approximations to the set of reachable program stores. In contrast, the backwards type analysis reviewed in this section propagates type information in the reverse direction of program execution and computes lower approximations to the set of program stores from which the execution will not violate any type assertions.

6 Directions for research on backwards analysis

6.1 Backwards analysis and module interaction

When reasoning about module interaction it can be advantageous to reverse the traditional deductive approach to abstract interpretation that is based on the abstract unfolding of abstract goals. In particular, [27] shows how abduction and abstraction can be combined to compute those properties that one module must satisfy to ensure that its composition with another fulfils certain requirements. Abductive analysis can, for example, determine how an optimisation in one module depends on a predicate defined in another module. Abductive analysis is related to backwards analysis since abduction is the inverse of deduction in much the same way that relative pseudo-complement is the reverse of conjunction. This suggests that the relationship between backwards analysis and abductive analysis warrants further investigation.

6.2 Backwards analysis and unfolding

Automatic program specialisation is a reoccurring theme in logic program development and one important aspect of this is the control of polyvariance [58]. Too much polyvariance (too many versions of a predicate) can lead to code bloat whereas too little polyvariance (too few versions of a predicate) can impede program specialisation and thereby efficiency. Surprisingly few works have addressed the problem of relating polyvariance to the ensuing optimisations [58], but recent work [49] has suggested that backwards analysis can be applied to control polyvariance by inferring specialisation conditions. Backwards analysis then becomes a pre-processing step that precedes the goal-dependent analysis and determines the degree of unfolding. Specifically, if the specialisation conditions are satisfied by an (abstract) call in a goal-dependent analysis then the call will possibly lead to valuable optimisations, and therefore it should not be merged with calls that lead to a lower level of optimisation. The backwards analysis in effect provides a convenient separation of concerns in that it enables version generation decisions to be made prior to applying top-down analysis. This work generalises and refines earlier work on compile-time garbage collection [48] that presents a kind of *ad hoc* backwards analysis for deriving reuse conditions for Mercury [62]. These works, and in particular [49], show how backwards analysis can provide a useful separation of concerns: the backwards analysis infers specialisation conditions which are later used in version control. This is reminiscent of the separation of control from unfolding that arises in off-line binding-time analysis [6]. In fact one promising direction for research would be to investigate how termination inference can be adapted to infer conditions under which loops can be partially unfolded.

6.3 Backwards analysis and Hoare logic

Pedreschi and Ruggieri [55] develop a calculus of weakest pre-conditions and weakest liberal pre-conditions, the latter of which is essentially a reformulation of Hoare's logic. Weakest liberal pre-conditions are characterised as the greatest fixpoint of a co-continuous operator on the space of interpretations. The work is motivated by, among other things, the desire to infer the absence of ill-typed arithmetic. Interestingly, it has been recently shown [40] that backwards analysis not only infers sufficient pre-conditions but the weakest pre-conditions. On the practical side, it means that backwards analysis need not be applied if forwards analysis cannot verify that a given query satisfies the assertions. Conversely, if an initial query is not inferred by backwards analysis, then it follows that forwards analysis cannot infer that the query satisfies the assertions. More generally, the expressive power of any backwards analysis needs to be compared against that of the forwards analysis that it attempts to reverse.

6.4 Backwards analysis and domain refinement

Recent work in domain refinement [29] has shown that the problem of minimally enriching an abstract domain to make it condense reduces to the problem of making the domain complete with respect to unification. Specifically, the work shows that unification coincides with multiplicative conjunction in a quantale of (idempotent) substitutions and that elements in a complete refined (condensing) abstract domain can be expressed in terms of linear logic. The significance of this work for backwards analysis, is that it provides a pathway for synthesising condensing domains that are not necessarily downward-closed. This suggests that the framework of [39] needs to be revised to accommodate these domains.

6.5 Backwards analysis and transformation

Very recently Gallagher [22] has proposed program transformation as a tactic for realising backwards analysis in terms of forwards analysis. Assertions are realised with a meta-predicate $\mathrm{d}(G, P)$ which expresses the relationship between an initial goal G and a property P to be checked at some program point. The meta-predicate $\mathrm{d}(G,P)$ holds if there is a derivation starting from G leading to the program point. The transformed program defining the predicate d can be seen as a realisation of the resultants semantics [21]. Backwards analysis is performed by examining the meaning of d, which can be approximated using a standard forwards analysis, to deduce goals G that imply that the property P holds. This work is both promising and intriguing because it finesses the requirement of calculating a greatest fixpoint. One interesting line of enquiry would be to compare the expressive power of transformation – the pre-conditions its infers – against those deduced via a bespoke backwards analysis framework [39,44].

7 Concluding discussion

This paper has shown how four classic program analysis and program development problems can be reversed. Reversal is a laudable goal in program analysis because it transforms a goal-dependent, checking problem into a goal-independent, inference problem; the latter being more general than the former. Arguably the greatest strength of backwards analysis is its ease of automation: backwards analyses can be surprisingly simple to implement and efficient to apply, and goal-independence means that it can be applied without any programmer interaction. Programmers merely have to interpret the inferred results and inspect the program if the results do not match their expectations. Thus, although backwards analysis is not yet a mainstream technology in the analysis of logic programs, its benefits need to be carefully weighed when a particular program development problem is being considered.

Backwards analysis is a modern approach to the analysis of logic programs in the sense that it relies on ideas that have been developed comparatively recently within the context of domain refinement. Backwards analysis thus illustrates the value of foundational work in logic program development. It also demonstrates the benefits of developing programs within the context of logic programming: the elegance of the underlying semantics manifests itself in the simplicity of the analyses. In fact it is fair to say that if we have seen slightly further in program development, it is only because we stand on the shoulders of those who have developed the underpinning semantics and abstract interpretation techniques.

Acknowledgements Our work on backwards analysis has greatly benefited from discussions with Maurice Bruynooghe, Mike Codish, John Gallagher Samir Genaim, Roberto Giacobazzi, Bart Massey, Fred Mesnard, Germán Puebla, Francesca Scozzari to name but a few. We also thank the anonymous referees for their valuable comments. This work was supported, in part, by the Nuffield Foundation grant NAL/00478/G and the National Science Foundation grants CCR-0131862 and INT-0327760.

References

1. A. Aiken and T. K. Lakshman. Directional Type Checking of Logic Programs. In B. Le Charlier, editor, *Static Analysis Symposium*, volume 864 of *Lecture Notes in Computer Science*, pages 43–60. Springer-Verlag, 1994.
2. 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.
3. F. Benoy and A. King. Inferring Argument Size Relationships with CLP($\mathcal{R}$). In J. Gallagher, editor, *Logic Program Synthesis and Transformation (Selected Papers)*, volume 1207 of *Lecture Notes in Computer Science*, pages 204–224. Springer-Verlag, 1996.
4. J. Boye, W. Drabent, and J. Małuszyński. Declarative Diagnosis of Constraint Programs: an Assertion-based Approach. In *Proceedings of the Third International Workshop on Automated Debugging*, pages 123–141. University of Linköping Press, 1997.
5. A. Brodsky and Y. Sagiv. Inference of Monotonicity Constraints in Datalog Programs. In *Symposium on Principles of Database Systems*, pages 190–199. ACM Press, 1989.
6. M. Bruynooghe, M. Leuschel, and K. Sagonas. A Polyvariant Binding-Time Analysis for Off-line Partial Deduction. In C. Hankin, editor, *European Symposium on Programming*, volume 1381 of *Lecture Notes in Computer Science*, pages 27–41. Springer-Verlag, 1998.
7. M. Carlsson. Freeze, Indexing, and Other Implementation Issues in the WAM. In J.-L. Lassez, editor, *International Conference on Logic Programming*, pages 40–58. MIT Press, 1987.

8. M. Codish, M. Falaschi, and K. Marriott. Suspension Analyses for Concurrent Logic Programs. *Transactions on Programming Languages and Systems*, 16(3):649–686, 1994.
9. M. Codish and V. Lagoon. Type Dependencies for Logic Programs using ACI-unification. *Theoretical Computer Science*, 238:131–159, 2000.
10. M. Codish and C. Taboch. A Semantic Basis for the Termination Analysis of Logic Programs. *The Journal of Logic Programming*, 41(1):103–123, 1999.
11. C. Codognet, P. Codognet, and M. Corsini. Abstract Interpretation for Concurrent Logic Languages. In S. K. Debray and M. V. Hermenegildo, editors, *North American Conference on Logic Programming*, pages 215–232. MIT Press, 1990.
12. M. Comini, R. Gori, G. Levi, and P. Volpe. Abstract Interpretation based Verification of Logic Programs. *Electronic Notes of Theoretical Computer Science*, 30(1), 1999.
13. A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Type Analysis of Prolog using Type Graphs. *The Journal of Logic Programming*, 22(3):179–208, 1995.
14. P. Cousot and R. Cousot. Inductive Principles for Proving Invariance Properties of Programs. In *Tools and Notions for Program Construction*, pages 75–119. Cambridge University Press, 1982.
15. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. In *Symposium on Principles of Programming Languages*, pages 84–97. ACM Press, 1978.
16. S. K. Debray. QD-Janus: a Sequential Implementation of Janus in Prolog. *Software Practice and Experience*, 23(12):1337–1360, 1993.
17. 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, 1992.
18. P. Dyber. Inverse Image Analysis Generalises Strictness Analysis. *Information and Computation*, 90(2):194–216, 1991.
19. M. Falaschi, P. Hicks, and W. Winsborough. Demand Transformation Analysis for Concurrent Constraint Programs. *The Journal of Logic Programming*, 41(3):185–215, 2000.
20. M. Gabbrielli and R. Giacobazzi. Goal Independency and Call Patterns in the Analysis of Logic Programs. In *ACM Symposium on Applied Computing*, pages 394–399. ACM Press, 1994.
21. M. Gabbrielli, G. Levi, and M. C. Meo. Resultants Semantics for Prolog. *Journal of Logic and Computation*, 6(4):491–521, 1996.
22. J. P. Gallagher. A Program Transformation for Backwards Analysis of Logic Programs. In M. Bruynooghe, editor, *Pre-proceedings of the International Symposium on Logic-based Program Synthesis and Transformation*, volume CW 365 of *Katholieke Universiteit Leuven, Technical Report*, pages 113–122, 2003.
23. J. P. Gallagher and G. Puebla. Abstract Interpretation over Non-deterministic Finite Tree Automata for Set-based Analysis of Logic Programs. In S. Krishnamurthi and C. R. Ramakrishnan, editors, *Practical Aspects of Declarative Languages*, volume 2257 of *Lecture Notes in Computer Science*, pages 243–261. Springer-Verlag, 2002.
24. S. Genaim and M. Codish. Inferring Termination Conditions for Logic Programs using Backwards Analysis. In R. Nieuwenhuis and A. Voronkov, editors, *International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 2250 of *Lecture Notes in Artificial Intelligence*,

pages 681–690. Springer-Verlag, 2001. Technical report version available at http://www.cs.bgu.ac.il/~mcodish/Papers/Pages/lpar01.html.
25. S. Genaim, M. Codish, J. P. Gallagher, and V. Lagoon. Combining Norms to Prove Termination. In A. Cortesi, editor, *Verification, Model Checking and Abstract Interpretation*, volume 2294 of *Lecture Notes in Computer Science*, pages 126–138. Springer-Verlag, 2002.
26. 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.
27. R. Giacobazzi. Abductive Analysis of Modular Logic Programs. *Journal of Logic and Computation*, 8(4):457–484, 1998.
28. 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.
29. R. Giacobazzi, F. Ranzato, and F. Scozzari. Making Abstract Domains Condensing. *ACM Transactions on Computational Logic*, To appear.
30. R. Giacobazzi and F. Scozzari. A Logical Model for Relational Abstract Domains. *ACM Transactions on Programming Languages and Systems*, 20(5):1067–1109, 1998.
31. C. Hall and D. Wise. Generating Function Versions with Rational Strictness Patterns. *Science of Computer Programming*, 12:39–74, 1989.
32. M. Hanus. Compile-time Analysis of Nonlinear Constraints in CLP($\mathcal{R}$). *New Generation Computing*, 13(2):155–186, 1995.
33. A. Heaton, M. Abo-Zaed, M. Codish, and A. King. A Simple Polynomial Groundness Analysis for Logic Programs. *The Journal of Logic Programming*, 45(1–3):143–156, 2000.
34. S. Hoarau and F. Mesnard. Inferring and Compiling Termination for Constraint Logic Programs. In P. Flener, editor, *Logic-based Program Synthesis and Transformation (Selected Papers)*, volume 1559 of *Lecture Notes in Computer Science*, pages 240–254. Springer-Verlag, 1998.
35. C. A. R. Hoare, I. J. Hayes, J. He, C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, and B. Sufrin. Laws of Programming. *Communications of the ACM*, 30(8):672–686, 1987.
36. J. M. Howe and A. King. Abstracting Numeric Constraints with Boolean Functions. *Information Processing Letters*, 75(1–2):17–23, 2000.
37. J. M. Howe and A. King. Positive Boolean Functions as Multiheaded Clauses. In P. Codognet, editor, *International Conference on Logic Programming*, volume 2237 of *Lecture Notes in Computer Science*, pages 120–134. Springer-Verlag, 2001.
38. J. M. Howe and A. King. Efficient Groundness Analysis in Prolog. *Theory and Practice of Logic Programming*, 3(1):95–124, 2003.
39. A. King and L. Lu. A Backward Analysis for Constraint Logic Programs. *Theory and Practice of Logic Programming*, 2:517–547, 2002.
40. A. King and L. Lu. Forward versus Backward Verification of Logic Programs. In C. Palamidessi, editor, *International Conference on Logic Programming*, volume 2916 of *Lecture Notes in Computer Science*, pages 315–330. Springer-Verlag, 2003.

41. A. King and P. Soper. Schedule Analysis of Concurrent Logic Programs. In K. R. Apt, editor, *Joint International Conference and Symposium on Logic Programming*, pages 478–492. MIT Press, 1992.
42. B. Le Charlier, C. Leclére, S. Rossi, and A. Cortesi. Automatic Verification of Prolog Programs. *The Journal of Logic Programming*, 39(1–3):3–42, 1999.
43. N. Lindenstrauss, Y. Sagiv, and A. Serebrenik. Unfolding the Mystery of Mergesort. In N. E. Fuchs, editor, *Logic Program Synthesis and Transformation (Selected Papers)*, volume 1463 of *Lecture Notes in Computer Science*, pages 206–225. Springer-Verlag, 1997.
44. L. Lu and A. King. Backward Type Inference Generalises Type Checking. In M. V. Hermenegildo and G. Puebla, editors, *Static Analysis Symposium*, volume 2477 of *Lecture Notes in Computer Science*, pages 85–101. Springer-Verlag, 2002.
45. K. Marriott, M. García de la Banda, and M. V. Hermenegildo. Analyzing Logic Programs with Dynamic Scheduling. In *Principles of Programming Languages*, pages 240–254. ACM Press, 1994.
46. 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.
47. J. C. Martin and A. King. Generating Efficient, Terminating Logic Programs. In *Theory and Practice of Software Development*, volume 1214 of *Lecture Notes in Computer Science*, pages 273–284. Springer-Verlag, 1997.
48. N. Mazur, G. Janssens, and M. Bruynooghe. A Module Based Analysis for Memory Reuse in Mercury. In *Computational Logic*, volume 1861 of *Lecture Notes in Artificial Intelligence*, pages 1255–1269, 2000.
49. N. Mazur, G. Janssens, and V. Van Hoof. Collecting Potential Optimizations. In M. Leuschel, editor, *Logic-based Program Synthesis and Transformation*, volume 2664 of *Lecture Notes in Computer Science*, pages 109–110, 2002.
50. F. Mesnard. Inferring Left-terminating Classes of Queries for Constraint Logic Programs. In *Joint International Conference and Symposium on Logic Programming*, pages 7–21. MIT Press, 1996.
51. F. Mesnard and U. Neumerkel. Applying Static Analysis Techniques for Inferring Termination Conditions of Logic Programs. In *Static Analysis Symposium*, volume 2126 of *Lecture Notes in Computer Science*, pages 93–110. Springer-Verlag, 2001.
52. F. Mesnard, É. Payet, and U. Neumerkel. Detecting Optimal Termination Conditions of Logic Programs. In M. V. Hermenegildo and G. Puebla, editors, *Static Analysis Symposium*, volume 2477 of *Lecture Notes in Computer Science*, pages 509–526. Springer-Verlag, 2002.
53. F. Mesnard and S. Ruggieri. On Proving Left Termination of Constraint Logic Programs. *ACM Transactions on Computational Logic*, 4(2):207–259, 2003.
54. L. Naish. Negation and Quantifiers in NU-Prolog. In E. Y. Shapiro, editor, *International Conference on Logic Programming*, volume 225 of *Lecture Notes in Computer Science*, pages 624–634. Springer-Verlag, 1986.
55. D. Pedreschi and S. Ruggieri. Weakest Preconditions for Pure Prolog Programs. *Information Processing Letters*, 67(3):145–150, 1998.
56. G. Puebla, F. Bueno, and M. V. Hermenegildo. An Assertion Language for Constraint Logic Programs. In *Analysis and Visualization Tools for Constraint Programming*, volume 1870 of *Lecture Notes in Computer Science*, pages 23–61. Springer-Verlag, 2000.

57. G. Puebla, F. Bueno, and M. V. Hermenegildo. A Generic Preprocessor for Program Validation and Debugging. In *Analysis and Visualization Tools for Constraint Programming*, volume 1870 of *Lecture Notes in Computer Science*, pages 63–107. Springer-Verlag, 2000.
58. G. Puebla and M. V. Hermenegildo. Abstract Multiple Specialization and its Application to Program Parallelization. *The Journal of Logic Programming*, 41(2&3):279–316, 1999.
59. P. Rychlikowski and T. Truderung. Polymorphic Directional Types for Logic Programming. In *International Conference on Principles and Practice of Declarative Programming*, pages 61–72. ACM Press, 2001.
60. V. A. Saraswat. *Concurrent Constraint Programming.* MIT Press, 1993.
61. V. A. Saraswat, K. Kahn, and J. Levy. Janus: a Step Towards Distributed Constraint Programming. In *North American Conference on Logic Programming*, pages 431–446. MIT Press, 1990.
62. Z. Somogyi, F. Henderson, and T. Conway. The Execution Algorithm of Mercury, an Efficient Purely Declarative Logic Programming Language. *The Journal of Logic Programming*, 29(1–3):17–64, 1996.
63. L. Vielle. Recursive Query Processing: The Power of Logic. *Theoretical Computer Science*, 69(1):1–53, 1989.
64. P. Volpe. A First-Order Language for Expressing Aliasing and Type Properties of Logic Programs. *Science of Computer Programming*, 39(1):125–148, 2001.
65. P. Wadler and R. J. M. Hughes. Projections for Strictness Analysis. In *Functional Programming and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 385–407. Springer-Verlag, 1987.
66. E. Yardeni and E. Y. Shapiro. A Type System for Logic Programs. *The Journal of Logic Programming*, 10(2):125–153, 1991.