Computer Science at Kent

A Backward Analysis for Constraint Logic Programs

Andy King and Lunjin Lu

Technical Report No: 4-01
Date: December 2001

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

Abstract

One recurring problem in program development is that of understanding how to re-use code developed by a third party. In the context of (constraint) 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 either experiment with queries in an *ad hoc* fashion or trace the control-flow of the program (backward) to infer the modes in which a predicate must be called so as to avoid an instantiation error. This paper presents an abstract interpretation scheme that automates the latter technique. The analysis presented in this paper can infer moding properties which if satisfied by the initial query, come with the guarantee that the program and query can never generate any moding or instantiation errors. Other applications of the analysis are discussed. The paper explains how abstract domains with certain computational properties (they condense) can be used to trace control-flow backward (right-to-left) to infer useful properties of initial queries. A correctness argument is presented and an implementation is reported.

1 Introduction

The myth of the lonely logic programmer writing a program in isolation is just that: a myth. Applications (and application components) are usually implemented and maintained by a team. One consequence of this is a significant proportion of the program development effort is devoted to understanding code developed by another. One advantage of (constraint) logic programs for software development is that their declarative nature makes them less opaque than, say, C++ programs. One disadvantage of logic programs over C++ programs, however, is that the signature (argument types) of a predicate do not completely specify how the predicate should be invoked. In particular, a call to a predicate from an unexpected context may generate an error if an argument of the call is insufficiently instantiated (even if the program and query are well-typed). This is because logic programs contain builtins and calls to these builtins often impose moding requirements on the query. If the program is developed by another programmer, it may not be clear how to query a predicate so as to avoid an instantiation error. In these circumstances, the programmer will often resort to a trial and error tactic in their search for an initial call mode. This can be both frustrating and tedious and, of course, cannot guarantee coverage of all the program execution paths. This paper presents an analysis for inferring moding properties which, if satisfied by the initial query, ensure that the program does not generate instantiation errors. Of course, it does not mean that the inferred call has the form exactly intended by the original programmer – no analysis can do that – the analysis just recovers mode information. Nevertheless, this is a useful first step in understanding the code developed by another.

The problem of inferring initial queries which do not lead to instantiation errors is an instance of the more general problem of deducing how to call a program so that it conforms to some desired property, for example, calls to builtins do not error, the program terminates, or calls to builtins behave predictably. The backward analysis presented in this paper is designed to infer conditions on the query which, if satisfied, guarantee that resulting derivations satisfy a property such as one of those above. Specifically, the analysis framework can be instantiated to solve the following analysis problems:

- Builtins and library functions can behave unpredictably when called with infinite rational trees. For example, the query ?- X = X + X, Y is X will not terminate in SICStus Prolog because the arithmetic operator expects its input to be a finite tree rather than an infinite rational tree. Moreover, the standard term ordering of Prolog does not lift to rational trees, so the builtin sort can behave unpredictably when sorting rational trees. These problems (and related problems with builtins) motivate the use of dependency analysis for tracking which terms are definitely finite [3]. The basic idea is to describe the constraint $x = f(x_1, \ldots, x_n)$ by the Boolean function $x \Leftrightarrow \wedge_{i=1}^{n} x_i$ which encodes that x is bound to a finite tree iff each x_i is bound to a finite tree. Although not proposed in the context of backward analysis [3], the framework proposed in this paper can be instantiated with a finite tree dependency domain to infer finiteness properties on the query which, if satisfied, guarantee that builtins are not called with problematic arguments.

- Termination inference is the problem of inferring initial modes for a query that, if satisfied, ensure that a logic program terminates. This problem generalises termination checking which verifies program termination for a class of queries specified by a given mode. Termination inference dates back to [29] but it has been recently observed [17] that the missing link between termination checking and termination inference is backward analysis. A termination inference analyser is reported in [17] composed from two components: a standard termination checker [8] and the backward analysis described in this paper. The resulting analyser is similar to the cTI analyser of [30] – the main difference is its design as two existing black-box components which, according to [17], simplifies the formal justification and implementation.

- Mode analysis is useful for implementing ccp programs. In particular [12] 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. If the guards of the predicates are re-interpreted as moding requirements, then the backward mode analysis can infer sufficient conditions for avoiding deadlock under left-to-right scheduling. The analysis presented in this paper thus has applications outside program development.

To summarise, the analysis presented in this paper can deduce properties of the call which, if satisfied, guarantee that resulting derivations fulfill some desired property. The analysis is unusual in that it applies lower approximation (see 2.4.1) as well as upper approximation (see 2.3.1); it is formulated in terms of a greatest fixpoint calculation (see 2.4) as well as least fixpoint calculation (see 2.3); the analysis also imposes some unusual restrictions on the abstract domain (see 2.4.6).

1.1 Backward analysis

Backward analysis has been applied extensively in functional programming in, among other things, projection analysis [38], stream strictness analysis [22], inverse image analysis [14], *etc.* By reasoning about the context of a function application, these analyses can identify opportunities for eager evaluation that are missed by (forward) strictness analysis as proposed by [31]. Furthermore, backward reasoning on imperative programs dates back to the early days of static analysis [9]. By way of contrast, backward analysis has been rarely applied in logic programming. One notable exception is the demand analysis of [11]. This analysis infers the degree of instantiation necessary for the guards of a concurrent constraint program (ccp) to reduce. It is a local analysis that does not consider the possible suspension of body calls. This analysis detects those (uni-modal) predicates which can be implemented with specialised suspension machinery. A more elaborate backward analysis for ccp is presented by [15]. This demand analysis infers how much input is necessary for a procedure to generate a certain amount of output. This information is useful for adding synchronisation (ask) constraints to a procedure to delay execution and thereby increase grain size, and yet not introduce deadlock. (Section 7 provides more extensive and reflective review of the related work.)

1.2 Contributions

Our work is quite different. As far as we are aware, it is unique in that it focuses on the backward analysis of (constraint) logic programs with left-to-right scheduling. Specifically, our work makes the following practical and theoretical contributions:

- it shows how to compute an initial mode of a predicate which is safe in that if a query is at least as instantiated as the inferred mode, the execution is guaranteed to be free from instantiation errors. The modes inferred are often disjunctive, sometimes surprising and, for the small predicates that we verified by hand, appear to be optimal.

- it specifies a practical algorithm for calculating initial modes that is straightforward to implement in that it reduces to two bottom-up fixpoint calculations. Furthermore, this backward analysis problem cannot be solved with any existing abstract interpretation machinery.

- to the best our knowledge, it is the first time domains that are closed under Heyting completion [21], or equivalently are condensing [27], have been applied to backward analysis. Put another way, our work adds credence to the belief that condensation is an important property in the analysis of logic programs.

The final point requires some unpacking. Condensation was originally proposed in [26], though arguably the simplest statement of this property [27] is for downward closed domains such as *Pos* [1] and the *Pos*-like type dependency domains [7]. Suppose that $f : X \to X$ is an abstract operation on a downward closed domain X equipped with an operation $\wedge$ that mimics unification or constraint solving. X is condensing iff $x \wedge f(y) = f(x \wedge y)$ for all $x, y \in X$. Hence, if X is condensing, $x \wedge f(true) = f(x)$ where $true$ represents the weakest abstract constraint. More exactly, if $f(true)$ represents the result of the goal-independent analysis, and $f(x)$ the result of the goal-dependent one with an initial constraint x, then the equivalence $f(x) = x \wedge f(true)$ enables goal-dependent analysis to be performed in a goal-independent way without loss of precision. This, in turn, can simplify the implementation of an analyser [1]. Because of this, domain refinement machinery has been devised to enrich a domain with new elements to obtain the desired condensing property [21]. It turns out that it is always possible to systematically design a condensing domain for a given downward closed property [21][Theorem 8.2] by applying Heyting completion. Conversely, under some reasonable hypotheses, all condensing domains can be reconstructed by Heyting completion [21][Theorem 8.3]. One consequence of this is that condensing domains come equipped with a (pseudo-complement) operator and this turns out to be an operation that is important in backward analysis. To summarise, machinery has been developed to synthesise condensing domains and condensing domains provide operations suitable for backward analysis.

1.3 Organisation of the paper

The rest of the paper is structured as follows. Section 2 introduces the key ideas of the paper in an informal way through a worked example. Section 3 introduces the necessary preliminaries for the formal sections that follow. Section 4 presents an operational semantics for constraint logic programs with assertions in which the set of program states is augmented by a special error state. Section 5 develops a semantics which computes those initial states that cannot lead to the error state. The semantics defines a framework for backward analysis and formally argues correctness. Section 6 describes an instantiation of the framework for mode analysis. Section 7 reviews the related work and section 8 concludes. Much of the formal machinery is borrowed directly from [19, 21] and in particular the reader is referred to [19] for proofs of the semantic results stated in section 3 (albeit presented in a slightly different form). To aid continuity in the paper, the remaining proofs are relegated to appendix A.

2 Worked example

2.1 Basic components

This section informally presents an abstract interpretation scheme which infers how to query a given predicate so as to avoid run-time moding errors. In other words, the analysis deduces moding properties of the call that, if satisfied, guarantee that resulting derivations cannot encounter an instantiation error. To illustrate, consider the Quicksort program listed in the left column of figure 1. This is the first ingredient of the analysis: the input program. The second ingredient is an abstract domain which, in this case, is *Pos*. *Pos* is the domain of positive Boolean functions, that is, the set of functions $f : \{0,1\}^n \to \{0,1\}$ such that $f(1,\ldots,1) = 1$. Hence $x \vee y \in Pos$ since $1 \vee 1 = 1$ but $\neg x \notin Pos$ since $\neg 1 = 0$. *Pos* is augmented with the bottom element 0 with 1 being the top element. The domain is ordered by entailment $\models$ and, in this example, will be used to represent grounding dependencies.

Pos comes equipped with the logical operations: conjunction $\wedge$, disjunction $\vee$, implication $\Rightarrow$ (and thus bi-implication $\Leftrightarrow$). Conjunction is used to conjoin the information from different body atoms, while disjunction is used to combine the information from different clauses. Conjunction and disjunction, in turn, enable two projection operators to be defined: $\exists_x(f) = f[x \mapsto 0] \vee f[x \mapsto 1]$ and $\forall_x(f) = f'$ if $f' \in Pos$ otherwise $\forall_x(f) = 0$ where $f' = f[x \mapsto 0] \wedge f[x \mapsto 1]$. Note that although $f[x \mapsto 0] \vee f[x \mapsto 1] \in Pos$ for all $f \in Pos$ it does not follow that $f[x \mapsto 0] \wedge f[x \mapsto 1] \in Pos$ for all $f \in Pos$. Indeed, $(x \Leftarrow y)[x \mapsto 0] \wedge (x \Leftarrow y)[x \mapsto 1] = \neg y$. Both operators are used to project out the body variables that are not in the head of a clause. Specifically, these operators eliminate the variable x from the formula f. They are dual in the sense that $\forall_x(f) \models f \models \exists_x(f)$. These are the basic components of the analysis.

2.2 Normalisation and abstraction

The analysis components are assembled in two steps. The first is a bottom-up analysis for success patterns, that is, a bottom-up analysis which infers the groundness dependencies which are known to be created by each predicate regardless of the calling pattern. This step is a least fixpoint (lfp) calculation. The second step is a bottom-up analysis for input modes (the objective of the analysis). This step is a greatest fixpoint (gfp) computation. 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 centre column of figure 1. This program is then abstracted by replacing each Herbrand constraint $x = f(x_1, \ldots, x_n)$ with a formula $x \Leftrightarrow \wedge_{i=1}^{n} x_i$ that describes its grounding dependency. This gives the abstract program listed in the right column of figure 1. The formula 1 in the assertion represents *true* whereas the formulae g_i that appear in the abstract program are as follows:

$$\begin{array}{ll} g_1 = t_1 \wedge (t_2 \Leftrightarrow s) & g_4 = t_1 \Leftrightarrow (x \wedge xs) \wedge t_2 \Leftrightarrow (x \wedge l) \\ g_2 = t_1 \Leftrightarrow (m \wedge xs) \wedge t_3 \Leftrightarrow (m \wedge r) & g_5 = t_1 \Leftrightarrow (x \wedge xs) \wedge t_2 \Leftrightarrow (x \wedge h) \\ g_3 = t_1 \wedge t_2 \wedge t_3 & g_6 = m \wedge x \end{array}$$

Builtins that occur in the source, such as the tests $=<$ and $>$, are handled by augmenting the abstract program with fresh predicates, $=<'$ and $>'$, which express the grounding behaviour of the builtins. The $\diamond$ symbol separates an assertion (the required mode) from another *Pos* formula describing the grounding behaviour of a successful call to the builtin (the success mode). For example, the formula g_6 left of $\diamond$ in the $=<'$ clause asserts that the $=<$ test will error if its first two arguments are not ground, whereas the g_6 right of $\diamond$ describes the state that holds if the test succeeds. These formulae do not coincide for all builtins (see Table 1). For quicksort, the only non-trivial assertions arise from builtins. This would change if the programmer introduced assertions for verification [32].

2.3 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. Renaming and equality of formulae induce an equivalence between success patterns which is needed to detect the fixpoint. The patterns $\langle p(u, w, v), u \wedge (w \Leftrightarrow v)\rangle$ 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.

```
qs([], s, s).
qs([m|xs], s, t) :-
    pt(xs,m,l,h),
    qs(l, s, [m|r]),
    qs(h, r, t).

pt([], _, [], []).
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).
```

```
qs(t1,s,t2) :-
    t1 = [], t2 = s.
qs(t1,s,t) :-
    t1 = [m|xs],
    t3 = [m|r],
    pt(xs,m,l,h),
    qs(l,s,t3),
    qs(h,r,t).

pt(t1,_,t2,t3) :-
    t1 = [],
    t2 = [], t3 = [].
pt(t1,m,t2,h) :-
    t1 = [x|xs],
    t2 = [x|l],
    m =< x
    pt(xs,m,l,h).
pt(t1,m,l,t2) :-
    t1 = [x|xs],
    t2 = [x|h],
    m > x,
    pt(xs,m,l,h).
```

```
qs(t1,s,t2) :-
    1 ◇ g1.
qs(t1,s,t) :-
    1 ◇ g2,
    pt(xs,m,l,h),
    qs(l,s,t3),
    qs(h,r,t).

pt(t1,_,t2,t3) :-
    1 ◇ g3.
pt(t1,m,t2,h) :-
    1 ◇ g4,
    =<'(m, x),
    pt(xs,m,l,h).
pt(t1,m,l,t2) :-
    1 ◇ g5,
    >'(m, x),
    pt(xs,m,l,h).

=<'(m, x) :- g6 ◇ g6.

>'(m, x) :- g6 ◇ g6.
```

Figure 1: Quicksort: raw, normalised and abstracted

2.3.1 Upper approximation of success patterns

A success pattern records an inter-argument groundness dependency that describes the binding effects of executing a predicate. If $\langle p(\vec{x}), f\rangle$ correctly describes the predicate p, and g holds whenever f holds, then $\langle p(\vec{x}), g\rangle$ also correctly describes p. Success patterns can thus be approximated from *above* without compromising correctness.

Iteration is performed in a bottom-up fashion and commences with $F_0 = \emptyset$. F_{j+1} is computed from F_j by considering each clause $p(\vec{x}) \leftarrow d \diamond f, p_1(\vec{x}_1), \ldots, p_n(\vec{x}_n)$ in turn. Initially $F_{j+1} = \emptyset$. The success pattern formulae f_i for the n body atoms are conjoined with f to obtain $g = f \wedge \wedge_{i=1}^{n} f_i$. Variables not present in $p(\vec{x})$, Y say, are then eliminated from g by computing $g' = \exists_Y(g)$ (weakening g) where $\exists_{\{y_1 \ldots y_n\}}(g) = \exists_{y_1}(\ldots \exists_{y_n}(g))$. Weakening g does not compromise correctness because success patterns can be safety approximated from above.

2.3.2 Weakening upper approximations

If F_{j+1} already contains a pattern of the form $\langle p(\vec{x}), g''\rangle$, then this pattern is replaced with $\langle p(\vec{x}), g' \vee g''\rangle$, otherwise F_{j+1} is revised to include $\langle p(\vec{x}), g'\rangle$. Thus the success patterns become progressively weaker on each iteration. Again, correctness is preserved because success patterns can be safety approximated from above.

2.3.3 Least fixpoint calculation for Quicksort

For brevity, let $\vec{u} = \langle x_1, x_2\rangle$, $\vec{v} = \langle x_1, x_2, x_3\rangle$ and $\vec{w} = \langle x_1, x_2, x_3, x_4\rangle$. Then the lfp for the abstracted Quicksort program is obtained (and checked) in the following 3 iterations:

$$F_1 = \left\{ \begin{array}{c} \langle \mathtt{qs}(\vec{v}), x_1 \wedge (x_2 \Leftrightarrow x_3)\rangle \\ \langle \mathtt{pt}(\vec{w}), x_1 \wedge x_3 \wedge x_4\rangle \\ \langle =<'(\vec{u}), x_1 \wedge x_2\rangle \\ \langle >'(\vec{u}), x_1 \wedge x_2\rangle \end{array} \right\} \quad F_2 = \left\{ \begin{array}{c} \langle \mathtt{qs}(\vec{v}), x_2 \Leftrightarrow (x_1 \wedge x_3)\rangle \\ \langle \mathtt{pt}(\vec{w}), x_1 \wedge x_3 \wedge x_4\rangle \\ \langle =<'(\vec{u}), x_1 \wedge x_2\rangle \\ \langle >'(\vec{u}), x_1 \wedge x_2\rangle \end{array} \right\}$$

Finally, $F_3 = F_2$. The space of success patterns forms a complete lattice which ensures that a lfp (a most precision solution) 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 iteration commences with the bottom element $F_0 = \emptyset$.

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.

2.4 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 violate 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). One starts with assuming no call causes an error and then checks this assumption 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).

2.4.1 Lower approximation of safe call patterns

Iteration commences with $D_0 = \{\langle p(\vec{x}), 1\rangle \mid p \in \Pi\}$ where Π is the set of predicate symbols occurring in the program. 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 $\langle p(\vec{x}), g\rangle$ correctly describes some safe call patterns of p, and g holds whenever f holds, then $\langle p(\vec{x}), f\rangle$ 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 considering each $p(\vec{x}) \leftarrow d \diamond f, p_1(\vec{x}_1), \ldots, p_n(\vec{x}_n)$ in turn and calculating a formula that characterises its safe calling modes. Initially set $D_{k+1} = D_k$. 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 $p_i(\vec{x}_i)$ in the previously computed lfp and let d_i denote the call pattern formula for $p_i(\vec{x}_i)$ in D_k. Set $e_{n+1} = 1$ 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 $p_i(\vec{x}_i), \ldots, p_n(\vec{x}_n)$.

2.4.2 Intuition and explanation

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

The basic inductive step in the analysis is to compute an e_i which ensures that $p_i(\vec{x}_i), \ldots, p_n(\vec{x}_n)$ does not error, given d_i and e_{i+1} which respectively ensure that $p_i(\vec{x}_i)$ and $p_{i+1}(\vec{x}_{i+1}), \ldots, p_n(\vec{x}_n)$ do not error. This step translates a demand after the call to $p_i(\vec{x}_i)$ into a demand before the call to $p_i(\vec{x}_i)$. The tactic is to set $e_{n+1} = 1$ 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 backward across the clause. Any calling mode is safe for the empty goal and hence $e_{n+1} = 1$. Note that $e_n = d_n \wedge (f_n \Rightarrow e_{n+1}) = d_n \wedge (\neg f_n \vee 1) = d_n$. Hence e_n represents a safe calling mode for the goal $p_n(\vec{x}_n)$.

Observe that e_i should not be larger than d_i, otherwise an error may occur while executing $p_i(\vec{x}_i)$. Observe too that if $p_i(\vec{x}_i), \ldots, p_n(\vec{x}_n)$ is called with a mode described by d_i, then $p_{i+1}(\vec{x}_{i+1}), \ldots, p_n(\vec{x}_n)$ is called with a mode described by $(d_i \wedge f_i)$ since f_i describes the success patterns of $p_i(\vec{x}_i)$. 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, $((d_i \wedge f_i) \Rightarrow e_{i+1}) = \vee\{f \in Pos \mid (d_i \wedge f_i) \wedge f \models e_{i+1}\}$.

Combining the requirements to satisfy $p_i(\vec{x}_i)$ and then $p_{i+1}(\vec{x}_{i+1}), \ldots, p_n(\vec{x}_n)$, gives $e_i = d_i \wedge ((d_i \wedge f_i) \Rightarrow e_{i+1})$ which reduces to $e_i = d_i \wedge (f_i \Rightarrow e_{i+1})$ and corresponds to the tactic used in the basic inductive step.

2.4.3 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 $e_i = 0$ 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 pseudo-complement operator. The pseudo-complement operator (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 did not possess a 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 [1] which does not possess a pseudo-complement. Def is the sub-class of Pos that is definite [1]. 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 0. Def can thus represent the grounding dependencies $x \wedge y$, x, $x \Leftrightarrow y$, y, $x \Leftarrow y$, $x \Rightarrow y$, 0 and 1 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 $e_i = 0$ would loose 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 $p_i(\vec{x}_i), \ldots, p_n(\vec{x}_n)$, 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 $p_i(\vec{x}_i)$ nor the sub-goal $p_{i+1}(\vec{x}_{i+1}), \ldots, p_n(\vec{x}_n)$ 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 $p_i(\vec{x}_i), \ldots, p_n(\vec{x}_n)$ 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 $p(\vec{x}) \leftarrow d \diamond f, p_1(\vec{x}_1), \ldots, p_n(\vec{x}_n)$.

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. Table 1 details how to abstract various builtins for groundness for a declarative subset of ISO Prolog.

2.4.4 Strengthening lower approximations

Variables not present in $p(\vec{x})$, Y say, are then eliminated by $g' = \forall_Y(g)$ (*strengthening* g) where $\forall_{\{y_1 \ldots y_n\}}(g) = \forall_{y_1}(\ldots \forall_{y_n}(g))$. A safe calling mode for this particular clause is then given by g'. Eliminating variables from g by strengthening g is unusual and initially appears strange. Recall, however, that call patterns can be approximated from below without compromising correctness (but not from above). In particular the standard projection tactic of computing $\exists_{\{y_1 \ldots y_n\}}(g)$ would result in an upper approximation of g that possibly describes a *larger* set of concrete call patterns which would be incorrect. The direction of approximation thus dictates that eliminating the variables Y from g must strengthen g. Indeed, g holds whenever $\forall_{y_i}(g)$ holds and therefore g holds whenever $\forall_{\{y_1 \ldots y_n\}}(g)$ holds as required.

D_{k+1} will contain a call pattern $\langle p(\vec{x}), g'' \rangle$ and, assuming $g' \wedge g'' \neq g''$, this is updated with

$\langle p(\vec{x}), g' \wedge g'' \rangle$. 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 the lfp is the most precise solution). Moreover, since the space of call patterns is finite, termination is assured. In fact, the scheme will converge onto the gfp since iteration commences with the top element $D_0 = \{\langle p(\vec{x}), 1\rangle \mid p \in \Pi\}$.

2.4.5 Greatest fixpoint calculation for Quicksort

Under this procedure Quicksort generates the following D_k sequence:

$$D_0 = \left\{ \begin{array}{r} \langle \mathtt{qs}(\vec{v}), 1\rangle \\ \langle \mathtt{pt}(\vec{w}), 1\rangle \\ \langle =<'(\vec{u}), 1\rangle \\ \langle >'(\vec{u}), 1\rangle \end{array} \right\} \quad D_1 = \left\{ \begin{array}{l} \langle \mathtt{qs}(\vec{v}), 1\rangle \\ \langle \mathtt{pt}(\vec{w}), 1\rangle \\ \langle =<'(\vec{u}), x_1 \wedge x_2\rangle \\ \langle >'(\vec{u}), x_1 \wedge x_2\rangle \end{array} \right\}$$

$$D_2 = \left\{ \begin{array}{l} \langle \mathtt{qs}(\vec{v}), 1\rangle \\ \langle \mathtt{pt}(\vec{w}), x_2 \wedge (x_1 \vee (x_3 \wedge x_4))\rangle \\ \langle =<'(\vec{u}), x_1 \wedge x_2\rangle \\ \langle >'(\vec{u}), x_1 \wedge x_2\rangle \end{array} \right\} \quad D_3 = \left\{ \begin{array}{l} \langle \mathtt{qs}(\vec{v}), x_1\rangle \\ \langle \mathtt{pt}(\vec{w}), x_2 \wedge (x_1 \vee (x_3 \wedge x_4))\rangle \\ \langle =<'(\vec{u}), x_1 \wedge x_2\rangle \\ \langle >'(\vec{u}), x_1 \wedge x_2\rangle \end{array} \right\}$$

These calculations are non-trivial so consider how D_2 is obtained from D_1 by applying the clause $\mathtt{pt}(t_1, m, t_2, h) :- 1 \diamond g_4, =<'(m, x), \mathtt{pt}(xs, m, l, h)$. The following e_i and g formulae are generated:

$$\begin{array}{ll} e_3 = & 1 \\ e_2 = & 1 \wedge ((xs \wedge l \wedge h) \Rightarrow 1) \;=\; 1 \\ e_1 = & (m \wedge x) \wedge ((m \wedge x) \Rightarrow 1) = \; m \wedge x \\ g = & 1 \wedge (((t_1 \Leftrightarrow x \wedge xs) \wedge (t_2 \Leftrightarrow x \wedge l)) \Rightarrow (m \wedge x)) \end{array}$$

To characterise those $\mathtt{pt}(t_1, m, t_2, h)$ calls which are safe, it is necessary to compute a function g' on the variables t_1, m, t_2, 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 $\mathtt{pt}(t_1, m, t_2, h)$) to strengthen g obtain a function g' such that g holds whenever g' holds. This is accomplished by calculating $g' = \forall_l \forall_{xs} \forall_x(g)$. First consider the computation of $\forall_x(g)$:

$$\begin{aligned} g[x \mapsto 0] &= (((t_1 \Leftrightarrow x \wedge xs) \wedge (t_2 \Leftrightarrow x \wedge l)) \Rightarrow (m \wedge x))[x \mapsto 0] \\ &= ((t_1 \Leftrightarrow 0 \wedge xs) \wedge (t_2 \Leftrightarrow 0 \wedge l)) \Rightarrow (m \wedge 0) \\ &= (\neg t_1 \wedge \neg t_2) \Rightarrow 0 \\ &= t_1 \vee t_2 \end{aligned}$$

$$\begin{aligned} g[x \mapsto 1] &= (((t_1 \Leftrightarrow x \wedge xs) \wedge (t_2 \Leftrightarrow x \wedge l)) \Rightarrow (m \wedge x))[x \mapsto 1] \\ &= ((t_1 \Leftrightarrow xs) \wedge (t_2 \Leftrightarrow l)) \Rightarrow m \end{aligned}$$

Since $g[x \mapsto 0] \wedge g[x \mapsto 1] \in Pos$ it follows that:

$$\forall_x(g) = (((t_1 \Leftrightarrow xs) \wedge (t_2 \Leftrightarrow l)) \Rightarrow m) \wedge (t_1 \vee t_2)$$

(otherwise $\forall_x(g)$ would be set to 0). Eliminating the other variables in a similar way we obtain:

$$\begin{aligned} \forall_{xs} \forall_x(g) &= ((t_2 \Leftrightarrow l) \Rightarrow m) \wedge (t_1 \vee t_2) \\ g' = \forall_l \forall_{xs} \forall_x(g) &= m \wedge (t_1 \vee t_2) \end{aligned}$$

Observe that if $\forall_l \forall_{xs} \forall_x(g)$ 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 $\langle \texttt{pt}(\vec{w}), x_2 \wedge (x_1 \vee x_3)\rangle$. Similarly the first and third clauses contribute the patterns $\langle \texttt{pt}(\vec{w}), 1\rangle$ and $\langle \texttt{pt}(\vec{w}) \leftarrow x_2 \wedge (x_1 \vee x_4)\rangle$. Observe also that

$$1 \wedge (x_2 \wedge (x_1 \vee x_3)) \wedge (x_2 \wedge (x_1 \vee x_4)) = x_2 \wedge (x_1 \vee (x_3 \wedge x_4))$$

which gives the final call pattern formula for $\texttt{pt}(\vec{w})$ 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 $\mathrm{pt}(\vec{w})$ 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.

2.4.6 Restrictions posed by the framework

The chief computational requirement of the analysis is that the input domain is equipped with a pseudo-complement operation. As already mentioned, it is always possible to systematically design a domain with this operator [21] and any domain that is known to be condensing (see section 1.2) comes equipped with this operator. Currently, however, there are only a few domains with a pseudo-complement. Indeed, the domain described in [7] appears to be unique in that it is the only type domain that is condensing. This is the main limitation of the backward analysis described in this paper.

Pos is downward-closed in the sense that if a function f describes a substitutions, then f also describes all substitutions less general than the substitution. The type domain of [7] is also downward-closed. It does not follow, however, that a domain equipped with a pseudo-complement operation is necessarily downward-closed. Heyting completion, the domain refinement technique used to construct pseudo-complement, can be moved to linear implication [20], though the machinery is more complicated. However, it is likely, that in the short term tractable condensing domains will continue to be downward-closed. In fact, constructing tractable downward-closed condensing domains is a topic within itself.

3 Preliminaries

3.1 Basic Concepts

Sets and sequences Let $\mathbb{N}$ denote the set of non-negative integers. The powerset of S is denoted $\wp(S)$. The empty sequence is denoted ϵ and $S^\star$ denotes the set of (possibly empty) sequences whose elements are drawn from S. Sequence concatenation is denoted $\cdot$ and the length of a sequence s is $|s|$. Furthermore, let $s^0 = \epsilon$ and $s^n = s \cdot s^{n-1}$ where $n \in \mathbb{N}$. If $n \in \mathbb{N}$ and $s \in \mathbb{N}^\star$ then $\max(n \cdot s) = \max(n, \max(s))$ where $\max(\epsilon) = 0$.

Orderings A pre-order on a set S is a binary relation $\sqsubseteq$ that is reflexive and transitive. A partial order on a set S is a pre-order that is anti-symmetric. A poset $\langle S, \sqsubseteq\rangle$ is a partial order on a set S. If $\langle S, \sqsubseteq\rangle$ is a poset, then $C \subseteq S$ is a chain iff $a \sqsubseteq b$ or $b \sqsubseteq a$ for all $a, b \in C$. A meet semi-lattice $\langle L, \sqsubseteq, \sqcap\rangle$ is a poset $\langle L, \sqsubseteq\rangle$ such that the meet (greatest lower bound) $\sqcap\{x, y\}$ exists for all $x, y \in L$. A complete lattice is a poset $\langle L, \sqsubseteq\rangle$ such that the meet $\sqcap X$ and the join $\sqcup X$ (least upper bound) exist for all $X \subseteq L$. Top and bottom are respectively defined by $\top = \sqcap\emptyset$ and $\bot = \sqcup\emptyset$. A complete lattice is denoted $\langle L, \sqsubseteq, \sqcap, \sqcup, \top, \bot\rangle$. Let $\langle S, \sqsubseteq\rangle$ be a pre-order. If $X \subseteq S$ then $\Downarrow(X) = \{y \in S \mid \exists x \in X. y \sqsubseteq x\}$. If $x \in S$ then $\Downarrow(x) = \Downarrow(\{x\})$. The set of order-ideals of S, denoted

$\wp^{\downarrow}(S)$, is defined by $\wp^{\downarrow}(S) = \{X \subseteq S \mid X = \Downarrow(X)\}$. Observe that $\langle \wp^{\downarrow}(S), \subseteq, \cup, \cap, S, \emptyset\rangle$ is a complete lattice.

An algebraic structure is a pair $\langle S, \mathcal{Q}\rangle$ where S is a non-empty set and $\mathcal{Q}$ is collection of n-ary operations $f : S^n \to S$ where $n \in \mathbb{N}$. Let $\langle S, \sqsubseteq\rangle$ and $\langle S', \sqsubseteq'\rangle$ be posets and $\langle S, \mathcal{Q}\rangle$ and $\langle S', \mathcal{Q}'\rangle$ algebraic structures such that $\mathcal{Q} = \{f_i \mid i \in I\}$ and $\mathcal{Q}' = \{f'_i \mid i \in I\}$ for an index set I. Then $\alpha : S \to S'$ is a semi-morphism between $\langle S, \mathcal{Q}\rangle$ and $\langle S', \mathcal{Q}'\rangle$ iff $\alpha(f_i(s_1, \ldots, s_n)) \sqsubseteq f'_i(\alpha(s_1), \ldots, \alpha(s_n))$ for all $\langle s_1, \ldots, s_n\rangle \in S^n$ and $i \in I$.

Functions and fixpoints Let $f : A \to B$. Then $\text{dom}(f)$ denotes the domain of f and if $C \subseteq A$ then $f(C) = \{f(c) \mid c \in C\}$. Furthermore, $\text{cod}(f) = f(\text{dom}(f))$. Let $\langle L, \sqsubseteq, \sqcup, \sqcap\rangle$ and $\langle L', \sqsubseteq', \sqcup', \sqcap'\rangle$ be complete lattices. The map $f : L \to L'$ is additive iff $f(\sqcup X) = \sqcup' f(X)$ for all $X \subseteq L$; f is continuous iff $f(\sqcup C) = \sqcup' f(C)$ for all chains $C \subseteq L$; f is co-continuous iff $f(\sqcap C) = \sqcap' f(C)$ for all chains $C \subseteq L$ and f is monotonic iff $f(x) \sqsubseteq' f(y)$ for all $x \sqsubseteq y$. Let $x \sqsubseteq y$. If f is continuous then $f(y) = f(x \sqcup y) = \sqcup'\{f(x), f(y)\}$ and thus $f(x) \sqsubseteq' f(y)$. If f is co-continuous then $f(x) = f(x \sqcap y) = \sqcap'\{f(x), f(y)\}$ and thus $f(x) \sqsubseteq' f(y)$. Both continuity and co-continuity thus imply monotonicity. If $f : L \to L$, then f is idempotent iff $f(x) = f^2(x)$ for all $x \in L$ and f is extensive iff $x \sqsubseteq f(x)$ for all $x \in L$. The Knaster-Tarski theorem states that any monotone operator $f : L \to L$ on a complete lattice $\langle L, \sqsubseteq, \sqcup, \sqcap, \top, \bot\rangle$ admit both greatest and least fixpoints that are characterised by $\text{gfp}(f) = \sqcup\{x \in L \mid x \sqsubseteq f(x)\}$ and $\text{lfp}(f) = \sqcap\{x \in L \mid f(x) \sqsubseteq x\}$. If f is co-continuous then $\text{gfp}(f) = \sqcap_{n \in \mathbb{N}} f^n(\top)$ and dually if f is continuous then $\text{lfp}(f) = \sqcup_{n \in \mathbb{N}} f^n(\bot)$. $\{f^n(\top) \mid n \in \mathbb{N}\}$ and $\{f^n(\bot) \mid n \in \mathbb{N}\}$ are, respectively, the lower and upper Kleene iteration sequences of f.

Galois insertions and closure operators If $\langle S, \sqsubseteq\rangle$ and $\langle S', \sqsubseteq'\rangle$ are posets and $\alpha : S \to S'$ and $\gamma : S' \to S$ are monotonic maps such that $\forall x \in S.x \sqsubseteq \gamma(\alpha(x))$ and $\forall x' \in S'.\alpha(\gamma(x')) \sqsubseteq' x'$, then the quadruple $\langle S, \gamma, S', \alpha\rangle$ is a Galois connection between S and S'. In other words, α is the lower (or left) adjoint of γ and γ is the upper (or right) adjoint of α. If, in addition, $\forall x' \in S'.x' \sqsubseteq' \alpha(\gamma(x'))$, then $\langle S, \gamma, S', \alpha\rangle$ is a Galois insertion between S and S'. The operator $\rho : L \to L$ on a complete lattice $\langle L, \sqsubseteq\rangle$ is a closure operator iff ρ is monotonic, idempotent and extensive. The set of closure operators on L is denoted $uco(L)$. The image set $\rho(L)$ of a closure operator ρ is a complete lattice with respect to $\sqsubseteq$. A Galois insertion $\langle L, \gamma, L', \alpha\rangle$ between the complete lattices L and L' defines the closure operator $\rho = \gamma \circ \alpha$. Conversely, a closure operator $\rho : L \to L$ on the complete lattice $\langle L, \sqsubseteq, \sqcup\rangle$ defines the Galois insertion $\langle L, id, \rho(L), \rho\rangle$ where id denotes identity. Galois insertions and closure operators are thus isomorphic, though closure operators are typically more succinct and hence used in this paper.

Substitutions Let Sub denote the set of (idempotent) substitutions and let Ren denote the set of (bijective) renaming substitutions.

3.2 Cylindric constraint systems

Let V denote a (denumerable) universe of variables and let $\mathcal{C}$ denote a constraint system over V. An algebra $\langle \mathcal{C}, \trianglelefteq, \otimes, 1, \{\exists_x\}_{x \in V}, \{d_{x,y}\}_{x,y \in V}\rangle$ is a semi-cylindric constraint system iff $\langle \mathcal{C}, \trianglelefteq, \otimes\rangle$ is a meet semi-lattice with a top 1; $\exists_x$ is a family of (unary) cylindrification operations such that: $c \trianglelefteq \exists_x(c)$, $\exists_x(c) \trianglelefteq \exists_x(c')$ if $c \trianglelefteq c'$, $\exists_x(c \otimes \exists_x(c')) = \exists_x(c) \otimes \exists_x(c')$; and $d_{x,y}$ is a family of (constant) diagonalisation operations such that: $d_{x,x} = 1$, $d_{x,y} = \exists_z(d_{x,z} \otimes d_{z,y})$ and $d_{x,y} \otimes \exists_x(c \otimes d_{x,y}) \trianglelefteq c$ if $x \neq y$. Cylindrification captures the concept of projecting out a variable (and is useful in modeling variables that go out of scope) whereas diagonalisation captures the notion of an alias between two variables (and is useful in modeling parameter passing). (The reader is referred to [19] for further details on cylindric constraint systems and their application in abstract interpretation.)

Example 3.1 An equation e is a pair $(s = t)$ where s and t are terms. A finite conjunction of equations is denoted E and Eqn denotes the set of finite conjunctions of equations. Let $eqn(\theta) = \{x = t \mid x \mapsto t \in \theta\}$ and $unify(E) = \{\theta \in Sub \mid \forall(s = t) \in E.\theta(s) = \theta(t)\}$. Eqn is pre-ordered by entailment $E_1 \trianglelefteq E_2$ iff $unify(E_1) \subseteq unify(E_2)$ and quotiented by $E_1 \approx E_2$ iff $E_1 \trianglelefteq E_2$ and $E_2 \trianglelefteq E_1$. This gives the meet semi-lattice $\langle Eqn/{\approx}, \trianglelefteq, \otimes\rangle$ with a top 1 where conjunction is defined $[E_1]_\approx \otimes [E_2]_\approx = [E_1 \cup E_2]_\approx$ and $1 = [\emptyset]_\approx$. Let $mgu(E) = \{\theta \in unify(E) \mid \forall\kappa \in unify(E) \, . \, eqn(\kappa) \trianglelefteq eqn(\theta)\}$. Finally, let $d_{x,y} = [\{x = y\}]_\approx$ and define project out by $\exists_x([E]_\approx) = [eqn(\{y \mapsto t \in \theta \mid x \neq y\})]_\approx$ if $\theta \in mgu(E)$. Otherwise, if $mgu(E) = \emptyset$, define $\exists_x([E]_\approx) = [\{a = b\}]_\approx$ where a and b are distinct constant symbols. Then $\langle Eqn/{\approx}, \trianglelefteq, \otimes, 1, \{\exists_x\}_{x\in V}, \{d_{x,y}\}_{x,y\in V}\rangle$ is a semi-cylindric constraint system.

An algebra $\langle \mathcal{C}, \trianglelefteq, \oplus, \otimes, 1, 0, \{\exists_x\}_{x\in V}, \{d_{x,y}\}_{x,y\in V}\rangle$ that extends a semi-cylindric constraint system to a complete lattice $\langle \mathcal{C}, \trianglelefteq, \oplus, \otimes, 1, 0\rangle$ is a cylindric constraint system. A semi-cylindric constraint system can be lifted to a cylindric constraint system via a power-domain construction. In particular $\langle \wp^{\downarrow}(\mathcal{C}), \subseteq, \cup, \cap, \mathcal{C}, \emptyset, \{\exists'_x\}_{x\in V}, \{d'_{x,y}\}_{x,y\in V}\rangle$ is a cylindric constraint system where $\exists'_x(C) = \Downarrow(\{\exists_x(c) \mid c \in C\})$ and $d'_{x,y} = \Downarrow(d_{x,y})$.

Example 3.2 *The semi-cylindric system of example 3.1 can be lifted to the cylindric system* $\langle \wp^{\downarrow}(Eqn), \subseteq, \cup, \cap, Eqn, \emptyset, \exists', d'\rangle$ *where* $\exists'_x(C) = \Downarrow(\{\exists_x(c) | c \in C\})$ *and* $d'_{x,y} = \Downarrow(d_{x,y})$.

In the sequel, unless otherwise stated, all constraint systems considered are over the same V and thus a cylindric constraint system will be simply denoted $\langle \mathcal{C}, \trianglelefteq, \oplus, \otimes, 1, 0, \exists, d\rangle$. Let $\mathrm{var}(o)$ denote the set of the variables in the syntactic object o and let $FV(c)$ denote the set of free variables in a constraint $c \in \mathcal{C}$, that is, $FV(c) = \{x \in \mathrm{var}(c) \mid \exists y \in V \, . \, c \neq \exists_x(c \otimes d_{x,y})\}$. Abbreviate project out by $\exists_{\{x_1,\ldots,x_n\}}(c) = \exists_{x_1}(\ldots(\exists_{x_n}(c)))$ and project onto by $\overline{\exists}_X(c) = \exists_{FV(c)\setminus X}(c)$. Let $d_{\vec{x},\vec{y}} = \otimes_{i=1}^n d_{x_i,y_i}$ where $\vec{x} = \langle x_1 \ldots x_n\rangle$ and $\vec{y} = \langle y_1 \ldots y_n\rangle$. If $c \in \mathcal{C}$ then let $\partial^{\vec{y}}_{\vec{x}}(c)$ denote the constraint obtained by replacing $\vec{x}$ with $\vec{y}$, that is, $\partial^{\vec{y}}_{\vec{x}}(c) = \exists_{\vec{z}}(\exists_{\vec{x}}(c \otimes d_{\vec{x},\vec{z}}) \otimes d_{\vec{z},\vec{y}})$ where $\mathrm{var}(\vec{z}) \cap (FV(c) \cup \mathrm{var}(\vec{x}) \cup \mathrm{var}(\vec{y})) = \emptyset$. Finally, if $C \subseteq \mathcal{C}$ then $\partial^{\vec{y}}_{\vec{x}}(C) = \{\partial^{\vec{y}}_{\vec{x}}(c) \mid c \in C\}$.

Example 3.3 Let X be a finite subset of V. The groundness domain $\langle EPos_X, \models, \curlyvee, \wedge, 1, 0\rangle$ [23] is a finite lattice where $EPos_X = \{0\} \cup \{\wedge F \mid F \subseteq X \cup E_X\}$, $E_X = \{x \Leftrightarrow y \mid x, y \in X\}$ and $f_1 \curlyvee f_2 = \wedge\{f \in EPos_X \mid f_1 \models f \wedge f_2 \models f\}$. $EPos_X$ is a cylindric constraint system with $d_{x,y} = (x \Leftrightarrow y)$ and $\exists_x(f) = f' \wedge f''$ where $f' = \wedge\{y \in Y \mid f \models y\}$, $f'' = \wedge\{e \in E_Y \mid f \models e\}$ and $Y = X \setminus \{x\}$.

Example 3.4 Let $Bool_X$ denote the Boolean functions over X. The dependency domain Pos_X [1] is defined by $Pos_X = \{0\} \cup \{f \in Bool_X \mid \wedge X \models f\}$. Henceforth Y abbreviates $\wedge Y$. The lattice $\langle Pos_X, \models, \vee, \wedge, 1, 0\rangle$ is finite and is a cylindric constraint system with $d_{x,y} = (x \Leftrightarrow y)$ and Schröder elimination defining $\exists_x(f) = f[x \mapsto 1] \vee f[x \mapsto 0]$.

3.3 Complete Heyting algebras

Let $\langle L, \sqsubseteq, \sqcap\rangle$ be a lattice with $x, y \in L$. The pseudo-complement of x relatively to y, if it exists, is a unique element $z \in L$ such that $x \sqcap w \sqsubseteq y$ iff $w \sqsubseteq z$. L is relatively pseudo-completed iff the pseudo-complement of x relative to y, denoted $x \rightarrow y$, exists for all $x, y \in L$. If L is also complete then it is a complete Heyting algebra (cHa). If $x, y \in L$ then $x \sqcap (x \rightarrow y) = x \sqcap y$. Furthermore, if $\langle L, \sqsubseteq, \sqcup, \sqcap \rangle$ is a cHa then $x \rightarrow y = \sqcup\{w \in L \mid x \sqcap w \sqsubseteq y\}$. The intuition behind the pseudo-complement of x relative to y is that it is the weakest element whose combination (meet) with x implies y. Interestingly pseudo-complement can be interpreted as the adjoint of conjunction. (The reader is referred to [37] for further details on complete Heyting algebras.) The following result [4][Chapter IX, Theorem 15] explains how a cHa depends on the additivity of meet.

Theorem 3.1 A complete lattice L is relatively pseudo-complemented iff $x \sqcap (\sqcup Y) = \sqcup\{x \sqcap y \mid y \in Y\}$ for all $x \in L$ and $Y \subseteq L$.

Example 3.5 Let $\{x, y\} \subseteq X$ and $f = (x \Leftrightarrow y)$. Then returning to $EPos_X$ of example 3.3, $f \wedge (\curlyvee\{x, y\}) = f \wedge (1) = f \neq (x \wedge y) = \curlyvee\{x \wedge y, x \wedge y\} = \curlyvee\{f \wedge x, f \wedge y\}$. Hence, by theorem 3.1, $EPos_X$ is not a cHa. Now consider Pos_X of example 3.4, and specifically let $f \in Pos_X$ and $G \subseteq Pos_X$. Since $\wedge$ distributes over $\vee$, it follows that $f \sqcap (\sqcup G) = \sqcup\{f \sqcap g \mid g \in G\}$, thus by theorem 3.1, Pos_X is a cHa. Similarly, $\cap$ distributes over $\cup$, and thus it follows by theorem 3.1 that $\wp^{\downarrow}(\mathcal{C})$ is also a cHa.

3.4 Constraint logic programs

Let Π denote a (finite) set of predicate symbols, let $Atom$ denote the set of (flat) atoms over Π with distinct arguments drawn from V, and let $\langle \mathcal{C}, \trianglelefteq, \oplus, \otimes, 1, 0, \exists, d \rangle$ be a semi-cylindric constraint system. The set of constrained atoms is defined by $Base^{\mathcal{C}} = \{p(\vec{x}) \mathrel{:\!-} c \mid p(\vec{x}) \in Atom \wedge c \in \mathcal{C}\}$. Let $FV(p(\vec{x}) \mathrel{:\!-} c) = \mathrm{var}(\vec{x}) \cup FV(c)$. Entailment $\trianglelefteq$ lifts to $Base^{\mathcal{C}}$ by $w_1 \trianglelefteq w_2$ iff $\overline{\exists}_{\vec{x}}(d_{\vec{x},\vec{x}_1} \otimes c_1) \trianglelefteq \overline{\exists}_{\vec{x}}(d_{\vec{x},\vec{x}_2} \otimes c_2)$ where $w_i = p(\vec{x}_i) \mathrel{:\!-} c_i$ and $\mathrm{var}(\vec{x}) \cap (FV(w_1) \cup FV(w_2)) = \emptyset$. This pre-order defines the equivalence relation $w_1 \approx w_2$ iff $w_1 \trianglelefteq w_2$ and $w_2 \trianglelefteq w_1$ to give a set of interpretations defined by $Int^{\mathcal{C}} = \wp(Base^{\mathcal{C}}/\approx)$. $Int^{\mathcal{C}}$ is ordered by $I_1 \sqsubseteq I_2$ iff for all $[w_1]_\approx \in I_1$ there exists $[w_2]_\approx \in I_2$ such that $w_1 \trianglelefteq w_2$. Let $\equiv$ denote the induced equivalence relation $I_1 \equiv I_2$ iff $I_1 \sqsubseteq I_2$ and $I_2 \sqsubseteq I_1$. $\langle Int^{\mathcal{C}}/\equiv, \sqsubseteq, \sqcup, \sqcap, \top, \bot \rangle$ is a complete lattice where $[I_1]_\equiv \sqcup [I_2]_\equiv = [I_1 \cup I_2]_\equiv$, $[I_1]_\equiv \sqcap [I_2]_\equiv = [\cup\{I \mid I \sqsubseteq I_1 \wedge I \sqsubseteq I_2\}]_\approx$, $\top = [\{[p(\vec{x}) \mathrel{:\!-} 1]_\approx \mid p(\vec{x}) \in Atom\}]_\equiv$ and $\bot = [\emptyset]_\equiv$.

A constraint logic program P over $\mathcal{C}$ is a finite set of clauses w of the form $w = h \mathrel{:\!-} c, g$ where $h \in Atom$, $c \in \mathcal{C}$, $g \in Goal$ and $Goal = Atom^\star$. The fixpoint semantics of P is defined in terms of an immediate consequences operator $\mathcal{F}_P^{\mathcal{C}}$.

Definition 3.1 Given a constraint logic program P over a semi-cylindric constraint system $\mathcal{C}$, the operator $\mathcal{F}_P^{\mathcal{C}} : Int^{\mathcal{C}} \rightarrow Int^{\mathcal{C}}$ is defined by:

$$\mathcal{F}_P^{\mathcal{C}}(I) = \left\{ [p(\vec{x}) \mathrel{:\!-} c']_\approx \;\middle|\; \begin{array}{ll} \exists & p(\vec{x}) \mathrel{:\!-} c, p_1(\vec{x}_1), \ldots, p_n(\vec{x}_n) \in P \quad . \\ \exists & \{[p_i(\vec{x}_i) \mathrel{:\!-} c_i]_\approx\}_{i=1}^n \subseteq I \quad . \\ & c' = c \otimes \otimes_{i=1}^n \overline{\exists}_{\vec{x}_i}(c_i) \end{array} \right\}$$

The operator $\mathcal{F}_P^{\mathcal{C}}$ lifts to $Int^{\mathcal{C}}/\equiv$ by $\mathcal{F}_P^{\mathcal{C}}([I]_\equiv) = [\mathcal{F}_P^{\mathcal{C}}(I)]_\equiv$. The lifting is monotonic and hence the fixpoint semantics for a program P over $\mathcal{C}$ exists and is denoted $\mathcal{F}^{\mathcal{C}}(P) = \mathrm{lfp}(\mathcal{F}_P^{\mathcal{C}})$. (The reader is referred to [5, 25] for further details on semantics and constraint logic programming.)

The operational semantics of P is defined in terms of a transition system $\rightarrow_P$ between states of the form $State = Goal \times \mathcal{C}$. To define the transition system, let $FV(\langle g; c \rangle) = \mathrm{var}(g) \cup FV(c)$ and $FV(h \mathrel{:\!-} c, g) = \mathrm{var}(h) \cup FV(c) \cup \mathrm{var}(g)$. To rename clauses with $\varphi \in Ren$ it is necessary to rename constraints with φ. Thus define $\varphi(h \mathrel{:\!-} c, g) = \varphi(h) \mathrel{:\!-} \partial_{\vec{x}}^{\varphi(\vec{x})}(c), \varphi(g)$. To rename apart from a syntactic object o, let $w \ll_o P$ indicate that there exists $w' \in P$ and $\varphi \in Ren$ such that $\mathrm{var}(\mathrm{cod}(\varphi)) \cap FV(w') = \emptyset$, $\varphi(w') = w$ and $FV(o) \cap FV(w) = \emptyset$.

Definition 3.2 Given a constraint logic program P over a semi-cylindric constraint system $\mathcal{C}$, $\rightarrow_P \subseteq State^2$ is the least relation such that:

$$s = \langle p(\vec{x}), g; c\rangle \rightarrow_P \langle g', g; c \otimes d_{\vec{x},\vec{x}'} \otimes c'\rangle$$

where $p(\vec{x}') \text{:-} c', g' \ll_s P$.

The operational semantics is specified by the transitive closure of the transition relation on (atomic) goals, that is, $\mathcal{O}^{\mathcal{C}}(P) = [\{[p(\vec{x}) \text{:-} c]_{\approx} \mid \langle p(\vec{x}); 1\rangle \rightarrow_P^{\star} \langle \epsilon; c\rangle\}]_{\equiv}$. The relationship between the operational and fixpoint semantics is stated below.

Theorem 3.2 $\mathcal{O}^{\mathcal{C}}(P) = \mathcal{F}^{\mathcal{C}}(P)$.

3.5 Abstract semantics for constraint logic programs

To apply abstraction techniques and finitely characterise $\mathcal{F}^{\mathcal{C}}(P)$, and thereby $\mathcal{O}^{\mathcal{C}}(P)$, the semi-cylindric domain $\mathcal{C}$ is replaced by the cHa $\wp^{\downarrow}(\mathcal{C})$ which is particularly amenable to approximation and backward reasoning.

If P is a constraint logic program over $\mathcal{C}$, then $\Downarrow(P) = \{h \text{:-} \Downarrow(c), g \mid h \text{:-} c, g \in P\}$. Furthermore, if $I \in Int^{\mathcal{C}}$, then let $\Downarrow([I]_{\equiv}) = [\{[p(\vec{x}) \text{:-} \Downarrow(c)]_{\approx} \mid [p(\vec{x}) \text{:-} c]_{\approx} \in I\}]_{\equiv}$. Note the overloading on $\approx$ and hence $\equiv$. The $\approx$ of $[p(\vec{x}) \text{:-} c]_{\approx}$ is induced by $\langle \mathcal{C}, \trianglelefteq\rangle$ whereas the $\approx$ of $[p(\vec{x}) \text{:-} \Downarrow(c)]_{\approx}$ is induced by $\langle \wp^{\downarrow}(\mathcal{C}), \subseteq\rangle$. The following proposition details the relationship between $\mathcal{F}^{\mathcal{C}}$ and $\mathcal{F}^{\wp^{\downarrow}(\mathcal{C})}$.

Proposition 3.1 $\Downarrow(\mathcal{F}^{\mathcal{C}}(P)) \sqsubseteq \mathcal{F}^{\wp^{\downarrow}(\mathcal{C})}(\Downarrow(P))$.

Let $\langle \mathcal{C}, \trianglelefteq, \oplus, \otimes, 1, 0, \exists, d\rangle$ denote a cylindric constraint system. If $\rho \in uco(\mathcal{C})$ then $\langle \rho(\mathcal{C}), \trianglelefteq, \otimes\rangle$ is a complete lattice. If ρ is additive, then $\langle \rho(\mathcal{C}), \trianglelefteq, \oplus, \otimes\rangle$ is a sub-lattice of $\langle \mathcal{C}, \trianglelefteq, \oplus, \otimes\rangle$. More generally, the join is denoted $\oplus'$. Observe that $\rho(\mathcal{C})$ has 1 and $\rho(0)$ for top and bottom and $c_1 \oplus c_2 \trianglelefteq \rho(c_1 \oplus c_2) = c_1 \oplus' c_2$ for all $c_1, c_2 \in \rho(\mathcal{C})$. A cylindric constraint system is obtained by augmenting $\rho(\mathcal{C})$ with cylindrification $\exists'_x$ and diagonalisation $d'_{x,y}$ operators. To abstract $\langle \mathcal{C}, \trianglelefteq, \oplus, \otimes, 1, 0, \exists, d\rangle$ safely with $\langle \rho(\mathcal{C}), \trianglelefteq, \oplus', \otimes, 1, \rho(0), \exists', d'\rangle$, ρ is required to be a semi-morphism [19] which additionally requires that $\rho(\exists_x(c)) \trianglelefteq \exists'_x(\rho(c))$ for all $c \in \mathcal{C}$ and $\rho(d_{x,y}) \trianglelefteq d'_{x,y}$ for all $x, y \in V$. In fact, these requirements turn out to be relatively weak conditions: most abstract domains come equipped with (abstract) operators to model projection and parameter passing.

Example 3.6 Consider the cylindric system $\langle \wp^{\downarrow}(Eqn), \subseteq, \cup, \cap, Eqn, \emptyset, \exists, d\rangle$ derived from the semi-cylindric system introduced in example 3.1. Let $Bool = Bool_V$ and $Pos = Pos_V$. Define $\alpha_{Pos} : \wp^{\downarrow}(Eqn) \rightarrow Pos$ by $\alpha_{Pos}(C) = \vee\{\alpha(\theta) \mid \theta \in mgu(E) \wedge E \in C\}$ and $\alpha(\theta) = \wedge\{x \Leftrightarrow \text{var}(t) \mid x \mapsto t \in \theta\}$. Also define $\gamma_{Pos} : Pos \rightarrow \wp^{\downarrow}(Eqn)$ by $\gamma_{Pos}(f) = \cup\{C \in \wp^{\downarrow}(Eqn) \mid \alpha_{Pos}(C) \models f\}$ and observe $\rho_{Pos} \in uco(\wp^{\downarrow}(Eqn))$ where $\rho_{Pos} = \gamma_{Pos} \circ \alpha_{Pos}$. To construct a semi-morphism, put $d'_{x,y} = \gamma_{Pos}(x \Leftrightarrow y)$ and $\exists'_x(C) = \gamma_{Pos}(f[x \mapsto 1] \vee f[x \mapsto 0])$ where $f = \alpha_{Pos}(C)$. Then $\rho_{Pos}(d_{x,y}) \subseteq d'_{x,y}$ and $\rho_{Pos}(\exists_x(C)) \subseteq \exists'_x(\rho_{Pos}(C))$ for all $C \in \wp^{\downarrow}(Eqn)$. Note that $C_1 \cap C_2 = \gamma_{Pos}(f_1 \wedge f_2)$ and $C_1 \oplus' C_2 = \gamma_{Pos}(f_1 \vee f_2)$ where $C_i = \gamma_{Pos}(f_i)$. Surprisingly $C_1 \oplus' C_2 \neq C_1 \cup C_2$ [16], as witnessed by $C_1 = \gamma_{Pos}(x)$ and $C_2 = \gamma_{Pos}(x \Leftrightarrow y)$ since $\{y = f(x,z)\} \notin C_1 \cup C_2$ whereas $\alpha_{Pos}(\{y = f(x,z)\}) = y \Leftrightarrow (x \wedge z) \models x \vee (x \Leftrightarrow y)$ so that $\{y = f(x,z)\} \in C_1 \oplus' C_2 = \gamma_{Pos}(x \vee (x \Leftrightarrow y))$. Nevertheless, ρ_{Pos} is a semi-morphism between $\langle \wp^{\downarrow}(Eqn), \subseteq, \cup, \cap, Eqn, \emptyset, \exists, d\rangle$ and $\langle \rho_{Pos}(\wp^{\downarrow}(Eqn)), \subseteq, \oplus', \cap, Eqn, \rho_{Pos}(\emptyset), \exists', d'\rangle$.

The operator ρ lifts to the complete lattice $Int^{\mathcal{C}}/\!\equiv$ by $\rho([I]_\equiv) = [\rho(I)]_\equiv$ where $\rho(I) = \{[p(\vec{x}) \mathrel{:\!-} \rho(c)]_\approx \mid [p(\vec{x}) \mathrel{:\!-} c]_\approx \in I\}$. Thus $\rho \in uco(Int^{\mathcal{C}}/\!\equiv)$. It is also useful to lift ρ to programs by $\rho(P) = \{h \mathrel{:\!-} \rho(c), g \mid h \mathrel{:\!-} c, g \in P\}$. The following result relates the fixpoint semantics of P to that of its abstraction $\rho(P)$.

Theorem 3.3 Let $\mathcal{C}$ be a cylindric constraint system. If $\rho \in uco(\mathcal{C})$ is a semi-morphism, then $\rho(\mathcal{F}^{\mathcal{C}}(P)) \sqsubseteq \mathcal{F}^{\mathrm{cod}(\rho)}(\rho(P))$.

Corollary 3.1 Let $\mathcal{C}$ be a semi-cylindric constraint system. If $\rho \in uco(\wp^{\downarrow}(\mathcal{C}))$ is a semi-morphism, then $\rho(\Downarrow(\mathcal{F}^{\mathcal{C}}(P))) \sqsubseteq \mathcal{F}^{\mathrm{cod}(\rho)}(\rho(\Downarrow(P)))$.

4 Constraint logic programs with assertions

We consider programs annotated with assertions [13]. When considering the operational semantics of a constraint logic program, it is natural to associate assertions with syntactic elements of the program such as predicates or the program points between body atoms. Without loss of generality, we decorate the neck of each clause with a set of constraints C that is interpreted as an assertion. When C is encountered, the store c is examined to determine whether $c \in C$ (modulo renaming). If $c \in C$ execution proceeds normally, otherwise an error state, denoted $\lozenge$, is entered and execution halts.

To formalise this idea, let $\mathcal{C}$ be a semi-cylindric constraint system and $\rho \in uco(\wp^{\downarrow}(\mathcal{C}))$. The assertion language (in whatever syntactic form it takes) is described by ρ. A clause of a constraint logic program over $\mathcal{C}$ with assertions over $\mathrm{cod}(\rho)$ then takes the form $h \mathrel{:\!-} C \diamond c, g$ where $h \in Atom$, $C \in \mathrm{cod}(\rho)$, $c \in \mathcal{C}$, $g \in Goal$ and $\diamond$ separates the assertion from the body of the clause. Notice that C is an order-ideal and thus downward closed. (C can thus represent disjunctions of constraints, but the semantics presented in this section should not be confused with a collecting semantics.) Note also that program transformation [32] can be used to express program point assertions in terms of our assertion language. To specify the behaviour of programs with assertions, let $State_\lozenge = State \cup \{\lozenge\}$, and let $\mathrm{CLP}(P) = \{h \mathrel{:\!-} c, g \mid h \mathrel{:\!-} C \diamond c, g \in P\}$. The following definition details how the operational semantics for the assertion language is realised in terms of projection, renaming and a test for inclusion.

Definition 4.1 Given a constraint logic program P over a semi-cylindric constraint system $\mathcal{C}$ with assertions over $\rho(\wp^{\downarrow}(\mathcal{C}))$, $\Rightarrow_P \subseteq State \times State_\lozenge$ is the least relation such that:

$$s = \langle p(\vec{x}), g; c\rangle \Rightarrow_P \begin{cases} \lozenge & \text{if} \quad p(\vec{x}') \mathrel{:\!-} C' \diamond c', g' \in P \\ & \wedge \quad \partial_{\vec{x}}^{\vec{x}'}(\overline{\exists}_{\vec{x}}(c)) \notin C' \\ \langle g', g; c \otimes d_{\vec{x},\vec{x}'} \otimes c'\rangle & \text{else if } p(\vec{x}') \mathrel{:\!-} c', g' \ll_s \mathrm{CLP}(P) \end{cases}$$

Recall that $p(\vec{x}') \mathrel{:\!-} c', g' \ll_s \mathrm{CLP}(P)$ ensures that the clause $p(\vec{x}') \mathrel{:\!-} c', g'$ does not share any variables with s. The operational semantics of P is then defined in terms of $\Rightarrow_P^\star$ as $\mathcal{A}^{\rho,\mathcal{C}}(P) = [\{[p(\vec{x}) \mathrel{:\!-} c]_\approx \mid \langle p(\vec{x}); 1\rangle \Rightarrow_P^\star \langle \epsilon; c\rangle\}]_\equiv$. The relationship between two operational semantics is stated in the following (trivial) result.

Proposition 4.1 $\mathcal{A}^{\rho,\mathcal{C}}(P) \sqsubseteq \mathcal{O}^{\mathcal{C}}(\mathrm{CLP}(P))$

Assertions are often used as interface between behaviour that is amenable to formalisation, for example as an operational semantics, and behaviour that is less tractable, for example, the semantics of a builtin [33]. More to the point, it is not always possible to infer the behaviour of a builtin from its definition, partly because builtins are often complicated and partly because builtins are often expressed in a language such as C. Our work requires assertions for each builtin in order to specify: its calling convention (for example, which arguments are required to be ground) and its success behaviour (for example, which arguments are grounded).

5 Backward fixpoint semantics for constraint logic programs with assertions

Let P be a constraint logic program over the semi-cylindric constraint system $\mathcal{C}$ with assertions over $\rho(\wp^{\downarrow}(\mathcal{C}))$. One natural and interesting question is whether the error state $\lozenge$ is reachable (or conversely not reachable) in P from an initial state $\langle p(\vec{x}); c\rangle$. For a given constraint logic program P with assertions, the backward fixpoint semantics presented in this section infers a (possibly empty) set of $c \in \mathcal{C}$ for which $\langle p(\vec{x}); c\rangle \not\Rightarrow_P^{\star} \lozenge$. The semantics formalises the informal backward analysis sketched in section 2.

For generality, the semantics is parameterised by $\mathcal{C}$ and ρ. The correctness argument requires ρ to be a semi-morphism between $\langle \wp^{\downarrow}(\mathcal{C}), \subseteq, \cup, \cap, \mathcal{C}, \emptyset, \exists, d\rangle$ and $\langle \rho(\wp^{\downarrow}(\mathcal{C})), \subseteq, \oplus', \cap, \mathcal{C}, \rho(\emptyset), \exists', d'\rangle$. Additionally, $\rho(\wp^{\downarrow}(\mathcal{C}))$ must be a cHa, that is, it must possess a pseudo-complement $\rightarrow'$. To explain, how pseudo-complement aids backward analysis consider the problem of inferring $c \in \mathcal{C}$ for which $\langle g; c\rangle \not\Rightarrow_P^{\star} \lozenge$ where $g = p_1(\vec{x}_1), \ldots, p_n(\vec{x}_n)$. Suppose $f_i \in \rho(\wp^{\downarrow}(\mathcal{C}))$ describes the success pattern for $p_i(\vec{x}_i)$, that is, if $\langle p_i(\vec{x}_i); 1\rangle \rightarrow_P^{\star} \langle \epsilon; c\rangle$ then $c \in f_i$. Moreover, suppose $d_i \in \rho(\wp^{\downarrow}(\mathcal{C}))$ approximates the initial call pattern for $p_i(\vec{x}_i)$, that is, if $c \in d_i$ then $\langle p_i(\vec{x}_i); c\rangle \not\Rightarrow_P^{\star} \lozenge$. Observe that $\langle p_{n-1}(\vec{x}_{n-1}), p_n(\vec{x}_n); c\rangle \not\Rightarrow_P^{\star} \lozenge$ if $c \in d_{n-1} \cap e$ and $e \cap (d_{n-1} \cap f_{n-1}) \subseteq d_n$. This follows since $\langle p_{n-1}(\vec{x}_{n-1}); c\rangle \not\Rightarrow_P^{\star} \lozenge$ because $c \in d_{n-1} \cap e \subseteq d_{n-1}$. Moreover, if $\langle p_{n-1}(\vec{x}_{n-1}), p_n(\vec{x}_n); c\rangle \Rightarrow_P^{\star} \langle p_n(\vec{x}_n); c'\rangle$ then $c' \in (d_{n-1} \cap e) \cap f_{n-1} \subseteq d_n$ and thus $\langle p_n(\vec{x}_n); c'\rangle \not\Rightarrow_P^{\star} \lozenge$. Putting $e = \rho(\emptyset)$ ensures $e \cap (d_{n-1} \cap f_{n-1}) \subseteq d_n$ and thereby achieves correctness. However, for precision, $d_{n-1} \cap e$ should be maximised. Since $\rho(\wp^{\downarrow}(\mathcal{C}))$ is a cHa, this reduces to assigning $e = \oplus'\{e' \in \rho(\wp^{\downarrow}(\mathcal{C})) \mid e' \cap (d_{n-1} \cap f_{n-1}) \subseteq d_n\} = (d_{n-1} \cap f_{n-1}) \rightarrow' d_n$. In general, without pseudo-complement, there is no *unique best* e that maximises precision (see example 5.1). The construction is generalised for $g = p_1(\vec{x}_1), \ldots, p_n(\vec{x}_n)$, by putting $e_n = \mathcal{C}$ and $e_i = d_i \cap ((d_i \cap f_i) \rightarrow' e_{i+1}) = d_i \cap (f_i \rightarrow' e_{i+1})$ for $1 \leq i < n$. Then $\langle g; c\rangle \not\Rightarrow_P^{\star} \lozenge$ if $c \in e_1$ as required. This iterated application of $\rightarrow'$ to propagate requirements right-to-left is the very essence of the backward analysis.

Example 5.1 Returning to examples 3.2–3.5, let $\alpha_{EPos}(C) = \curlyvee\{\alpha(\theta) \mid \theta \in mgu(E) \wedge E \in C\}$, $\gamma_{EPos}(f) = \cup\{C \in \wp^{\downarrow}(Eqn) \mid \alpha_{EPos}(C) \models f\}$ and $\rho_{EPos} = \gamma_{EPos} \circ \alpha_{EPos}$. Note that $C_1 \cap C_2 = \gamma_{EPos}(f_1 \wedge f_2)$ and $C_1 \oplus' C_2 = \gamma_{EPos}(f_1 \curlyvee f_2)$ where $C_i = \gamma_{EPos}(f_i)$. By defining $\exists'$ and d' in an analogous way to example 3.5, a semi-morphism ρ_{EPos} is constructed between $\langle \wp^{\downarrow}(Eqn), \subseteq, \cup, \cap, Eqn, \emptyset, \exists, d\rangle$ and $\langle \rho_{EPos}(\wp^{\downarrow}(Eqn)), \subseteq, \oplus', \cap, Eqn, \rho_{EPos}(\emptyset), \exists', d'\rangle$. Recall that $\rho_{EPos}(\wp^{\downarrow}(Eqn))$ is *not* a cHa. Now consider the problem of inferring an initial c for $\langle p_{n-1}(\vec{x}_{n-1}), p_n(\vec{x}_n); c\rangle$ within $\rho_{EPos}(\wp^{\downarrow}(Eqn))$. In particular let $d_{n-1} = \gamma_{EPos}(1)$, $f_{n-1} = \gamma_{EPos}(x \Leftrightarrow y)$ and $d_n = \gamma_{EPos}(x \wedge y)$. Then $e_j \cap (d_{n-1} \cap f_{n-1}) \subseteq d_n$ for $e_1 = \gamma_{EPos}(x)$ and $e_2 = \gamma_{EPos}(y)$ but $(e_1 \oplus' e_2) \cap (d_{n-1} \cap f_{n-1}) = \gamma_{EPos}((x \curlyvee y) \wedge 1 \wedge (x \Leftrightarrow y)) = \gamma_{EPos}(x \Leftrightarrow y) \not\subseteq \gamma_{EPos}(x \wedge y) = d_n$. Thus there is no unique e maximising precision.

Example 5.2 Identity $\rho_{\mathrm{id}} = \lambda x.x$ is the trivial semi-morphism between $\langle \wp^{\downarrow}(\mathcal{C}), \subseteq, \cup, \cap, \mathcal{C}, \emptyset, \exists, d\rangle$ and $\langle \wp^{\downarrow}(\mathcal{C}), \subseteq, \cup, \cap, \mathcal{C}, \emptyset, \exists, d\rangle$ where the pseudo-complement is given by $C_1 \rightarrow' C_2 = \{c \in \mathcal{C} \mid \forall c' \trianglelefteq c \,.\, c' \in C_1 \Rightarrow c' \in C_2\}$ [4].

Example 5.3 Recall that ρ_{Pos} is a semi-morphism between $\langle \wp^{\downarrow}(Eqn), \subseteq, \cup, \cap, Eqn, \emptyset, \exists, d\rangle$ and $\langle \rho_{Pos}(\wp^{\downarrow}(Eqn)), \subseteq, \oplus', \cap, Eqn, \rho_{Pos}(\emptyset), \exists', d'\rangle$. Although $\oplus' \neq \cup$, $\rho_{Pos}(\wp^{\downarrow}(Eqn))$ is a sub-cHa of $\wp^{\downarrow}(Eqn)$ with respect to $\cap$ and $\rightarrow'$ [35]. Moreover, pseudo-complement (intuitionistic implication) $\rightarrow'$ coincides with classic implication $\Rightarrow$ in the sense that $C_1 \rightarrow' C_2 = \gamma_{Pos}(f_1 \Rightarrow f_2)$ where $C_i = \gamma_{Pos}(f_i)$. This follows since $V \models f_2 \models (\neg f_1) \vee f_2$ and thus $f_1 \Rightarrow f_2 \in Pos$. Moreover, $f_1 \wedge f \models f_2$ iff $\models (f_1 \wedge f) \Rightarrow f_2$ iff $\models f \Rightarrow (\neg f_1) \vee f_2$ iff $f \models (\neg f_1) \vee f_2$. Hence $C_1 \rightarrow' C_2 = \oplus'\{C \in \rho_{Pos}(\wp^{\downarrow}(Eqn)) \mid C_1 \cap C \subseteq C_2\} = \gamma_{Pos}(f_1 \Rightarrow f_2)$. Thus $\rightarrow'$ is finitely computable for ρ_{Pos}. Finally note that $\neg$ and $\vee$ are defined on *Bool* rather than *Pos* since $\neg f \notin Pos$ iff $f \in Pos$.

Example 5.4 Now consider the problem of inferring an initial c for $\langle p_{n-1}(\vec{x}_{n-1}), p_n(\vec{x}_n); c\rangle$ within $\rho_{Pos}(\wp^{\downarrow}(Eqn))$. Analogous to example 5.1, let $d_{n-1} = \gamma_{Pos}(1)$, $f_{n-1} = \gamma_{Pos}(x \Leftrightarrow y)$ and $d_n = \gamma_{Pos}(x \wedge y)$. Then $e_j \cap (d_{n-1} \cap f_{n-1}) \subseteq d_n$ for $e_1 = \gamma_{Pos}(x)$ and $e_2 = \gamma_{Pos}(y)$ and $(e_1 \oplus' e_2) \cap (d_{n-1} \cap f_{n-1}) = \gamma_{Pos}((x \vee y) \wedge 1 \wedge (x \Leftrightarrow y)) = \gamma_{Pos}(x \wedge y) = d_n$. Thus there is a unique e maximising precision.

Since $\langle \rho(\wp^{\downarrow}(\mathcal{C})), \subseteq, \oplus', \cap, \mathcal{C}, \rho(\emptyset), \exists', d'\rangle$ is a cylindric constraint system, it follows that $e \subseteq \exists'_x(e)$ for all $e \in \rho(\wp^{\downarrow}(\mathcal{C}))$. A consequence of $e \subseteq \exists'_x(e)$ is that projection approximates from above. Approximation from above, however, is not entirely appropriate for backward analysis. In particular, observe that if $\langle g; c\rangle \not\Rightarrow^{\star}_P \lozenge$ for all $c \in e$, then it does not necessarily follow that $\langle g; c\rangle \not\Rightarrow^{\star}_P \lozenge$ for all $c \in \exists'_x(e)$. What is required is a dual notion of projection, say denoted $\forall'$, that approximates from below. Then $\langle g; c\rangle \not\Rightarrow^{\star}_P \lozenge$ for all $c \in \forall'_x(e)$. Although $\forall'$ is an abstract operator, the concept is defined for an arbitrary cylindric constraint system for generality.

Definition 5.1 If $\langle \mathcal{C}, \unlhd, \oplus, \otimes, 1, 0, \exists, d\rangle$ is a cylindric constraint system and $x \in V$ then $\forall_x : \mathcal{C} \rightarrow \mathcal{C}$ is a monotonic operator such that: $\exists_x(\forall_x(c)) \unlhd c$ and $c \unlhd \forall_x(\exists_x(c))$ for all $c \in \mathcal{C}$.

Recall that $\exists_x$ is monotonic and thus α is the lower adjoint of γ and γ is the upper adjoint of α. More exactly, it follows that $\forall_x$ can be automatically constructed from $\exists_x$ by $\forall_x(c) = \oplus\{c' \in \mathcal{C} \mid \exists_x(c') \unlhd c\}$. Observe that this ensures that $\forall_x$ is the most precise projection operator from below. For succinctness, define $\forall_{\{x_1,\ldots,x_n\}}(c) = \forall_{x_1}(\ldots(\forall_{x_n}(c)))$ and $\overline{\forall}_X(c) = \forall_{FV(c)\setminus X}(c)$.

Example 5.5 For ρ_{id}, let $\forall'_x(C) = \Downarrow(\{c \in C \mid \exists_x(c) = c\})$.

Example 5.6 For ρ_{Pos}, let $\forall'_x(C) = \gamma_{Pos}(f')$ if $f' \in Pos$ otherwise $\forall'_x(C) = \gamma_{Pos}(0)$ where $C = \gamma_{Pos}(f)$ and $f' = f[x \mapsto 0] \wedge f[x \mapsto 1]$. Observe that $\exists_x(f)[x \mapsto 0] \wedge \exists_x(f)[x \mapsto 1] = \exists_x(f)$ and hence $C \subseteq \exists'_x(C) = \forall'_x(\exists'_x(C))$ as required. Moreover, if $\forall'_x(C) = \gamma_{Pos}(0)$ then $\exists'_x(\forall'_x(C)) = \gamma_{Pos}(0) \subseteq C$. Otherwise $\exists_x(f[x \mapsto 0] \wedge f[x \mapsto 1]) = f[x \mapsto 0] \wedge f[x \mapsto 1] \models f$. Thus $\exists'_x(\forall'_x(C)) \subseteq C$ as required. Finally, note that $\forall'_x$ is finitely computable for ρ_{Pos}. For example if $C_i = \gamma_{Pos}(f_i)$, $f_1 = (x \Leftarrow y)$, $f_2 = (x \wedge y)$ and $f_3 = (x \vee y)$, then $\forall'_x(C_i) = \gamma_{Pos}(f'_i)$ where $f'_1 = 0$, $f'_2 = 0$ and $f'_3 = y$.

Backward analysis can now be formalised as follows.

Definition 5.2 Given a constraint logic program P over a semi-cylindric constraint system $\mathcal{C}$ with assertions over $\rho(\wp^{\downarrow}(\mathcal{C}))$, the operator $\mathcal{D}^{\rho,\mathcal{C}}_P : Int^{\mathrm{cod}(\rho)} \rightarrow Int^{\mathrm{cod}(\rho)}$ is defined by:

$$\mathcal{D}^{\rho,\mathcal{C}}_P(D) = \bigcup \left\{ E \;\middle|\; \begin{array}{lcr} \forall & [p(\vec{x}) \text{:-}\, e]_{\approx} \in E & . \\ \forall & p(\vec{x}) \text{:-}\, C \diamond c, p_1(\vec{x}_1), \ldots, p_n(\vec{x}_n) \in P & . \\ \exists & \{[p_i(\vec{x}_i) \text{:-}\, f_i]_{\approx}\}_{i=1}^n \subseteq F & . \\ \exists & \{[p_i(\vec{x}_i) \text{:-}\, d_i]_{\approx}\}_{i=1}^n \subseteq D & . \\ & e_{n+1} = \mathcal{C} \;\wedge\; e_i = d_i \cap (f_i \rightarrow' e_{i+1}) & \wedge \\ & e \subseteq \overline{\forall}'_{\vec{x}}(e_0) \;\wedge\; e_0 = C \cap (\rho(\Downarrow(c)) \rightarrow' e_1) & \end{array} \right\}$$

where $[F]_\equiv = \mathcal{F}^{\mathrm{cod}(\rho)}(\rho(\Downarrow(\mathrm{CLP}(P))))$.

Since $\mathcal{D}$ is parameterised by ρ and $\mathcal{C}$ it can interpreted as a backward analysis framework. $\mathcal{D}$ requires F, the success patterns of the program obtained by discarding the assertions, to be pre-computed. $\mathcal{D}$ considers each clause in the program in turn and calculates those states which ensure that the clause (and those it calls) will not violate an assertion. An abstraction which characterises these states is calculated by propagating requirements, represented as abstractions, right-to-left by repeated application of pseudo-complement. Projection from below then computes those states which, when restricted to the head variables, still ensure that no error arises in the clause (and those it calls). Repeated application of $\mathcal{D}$ yields a decreasing sequence of interpretations.

The operator $\mathcal{D}_P^{\rho,\mathcal{C}}$ lifts to $Int^{\mathrm{cod}(\rho)}/\equiv$ by $\mathcal{D}_P^{\rho,\mathcal{C}}([D]_\equiv) = [\mathcal{D}_P^{\rho,\mathcal{C}}(D)]_\equiv$. Since $\langle \mathrm{cod}(\rho), \sqsubseteq, \sqcup, \sqcap\rangle$ is a complete lattice, $\mathcal{D}_P^{\rho,\mathcal{C}}$ will possess a gfp if $\mathcal{D}_P^{\rho,\mathcal{C}}$ is monotonic. The existence of $\mathrm{gfp}(\mathcal{D}_P^{\rho,\mathcal{C}})$ is guaranteed by the following result since co-continuity implies monotonicity.

Proposition 5.1 $\mathcal{D}_P^{\rho,\mathcal{C}} : Int^{\mathrm{cod}(\rho)}/\equiv \to Int^{\mathrm{cod}(\rho)}/\equiv$ is co-continuous.

Since $\mathrm{gfp}(\mathcal{D}_P^{\rho,\mathcal{C}})$ exists, a backward fixpoint semantics can be defined $\mathcal{D}^{\rho,\mathcal{C}}(P) = \mathrm{gfp}(\mathcal{D}_P^{\rho,\mathcal{C}})$ and computed by lower Kleene iteration. To establish a connection between $\mathcal{D}^{\rho,\mathcal{C}}(P)$ and the operational semantics of P, it is useful to annotate the goals of a state with their depth in the computation tree. To formalise this idea $\Rightarrow_P$ is lifted to the annotated states $Conf_\Diamond = Conf \cup \{\Diamond\}$ where $Conf = Goal \times \mathcal{C} \times \mathbb{N}^\star$ to obtain the transition system $\Rrightarrow_P$.

Definition 5.3 Given a constraint logic program with assertions P over a semi-cylindric constraint system $\mathcal{C}$, $\Rrightarrow_P \subseteq Conf \times Conf_\Diamond$ is the least relation such that:

$$\langle p(\vec{x}), g; c; n \cdot h\rangle \Rrightarrow_P \begin{cases} \Diamond & \text{if } \langle p(\vec{x}), g; c\rangle \Rightarrow_P \Diamond \\ \langle g', g; c'; (n+1)^{|g'|} \cdot h\rangle & \text{if } \langle p(\vec{x}), g; c\rangle \Rightarrow_P \langle g', g; c'\rangle \end{cases}$$

The sequence $(n+1)^{|g'|}$ denotes $|g'|$ concatenations of $n+1$. The following result relates the depth of the goals of the annotated states to the iterates obtained by lower Kleene iteration. Informally, it says that if a constrained atom $p(\vec{x}) \mathbin{:\!-} e$ occurs in the interpretation obtained by applying $\mathcal{D}$ k times, and e characterises an initial state (in a certain sense), and the depth of the goals in a derivation starting at the initial state does not exceed k, then the derivation will not violate an assertion. The main safety theorem flows out of this result.

Lemma 5.1 Let $\langle p(\vec{y}); c''; 1\rangle = s_1 \Rrightarrow_P^\star s_n \Rrightarrow_P \Diamond$, $s_i = \langle g_i; c_i; h_i\rangle$ and $(\mathcal{D}_P^{\rho,\mathcal{C}})^k(\top) = [D_k]_\equiv$. If $\max(\{\max(h_i) \mid 1 \le i \le n\}) \le k$ and $[p(\vec{y}) \mathbin{:\!-} e]_\approx \in D_k$ then $\overline{\exists}_{\vec{y}}(c'') \notin \overline{\exists}_{\vec{y}}(e)$.

Theorem 5.1 If $\mathcal{D}^{\rho,\mathcal{C}}(P) = [D]_\equiv$, $[p(\vec{y}) \mathbin{:\!-} e]_\approx \in D$ and $c \in \overline{\exists}_{\vec{y}}(e)$ then $\langle p(\vec{y}); c\rangle \not\Rightarrow_P^\star \Diamond$.

6 Experimental evaluation

In order to evaluate the usefulness of the analysis framework presented in section 5, a backward *Pos* analyser has been constructed for inferring calling modes. The fixpoint component of the analyser is coded in SICStus Prolog 3.8.3. The domain operations are coded in C and are essentially the binary decision diagram (BDD) routines written by Armstrong and Schachte [1]. The analyser takes, as input, a program written in a declarative subset of ISO Prolog. It outputs a mode for each program predicate. The safety result of theorem 5.1 ensures that if a call to a predicate is at

builtin	*required mode*	*success mode*
$t_1 == t_2$, $t_1 \backslash== t_2$, t_1 @< t_2, t_1 @> t_2, t_1 @=< t_2, t_1 @>= t_2, $t_1 \backslash= t_2$, !, compound(t_1), display(t_1), listing, listing(t_1), nl, nonvar(t_1), print(t_1), portray_clause(t_1), read(t_1), repeat, true, var(t_1), write(t_1), writeq(t_1)	*true*	*true*
atom(t_1), atomic(t_1), compare(t_1, t_2, t_3), float(t_1), ground(t_1), integer(t_1), number(t_1)	*true*	f_1
length(t_1, t_2)	*true*	f_2
statistics(t_1, t_2)	*true*	g_1
abort, fail, false	*true*	*false*
keysort(t_1, t_2), sort(t_1, t_2)	f_1	g_2
tab(t_1), put(t_1)	f_1	f_1
t_1 is t_2	f_2	g_1
$t_1 =:= t_2$, $t_1 =\backslash= t_2$, $t_1 < t_2$, $t_1 > t_2$, $t_1 =< t_2$, $t_1 >= t_2$	g_1	g_1
arg(t_1, t_2, t_3)	g_1	g_3
name(t_1, t_2)	g_4	g_1
t_1 =.. t_2	g_4	g_2
functor(t_1, t_2, t_3)	g_5	g_6

Table 1: Abstracting builtins where $f_i = \wedge\text{var}(t_i)$, $g_1 = f_1 \wedge f_2$, $g_2 = f_1 \Leftrightarrow f_2$, $g_3 = f_1 \wedge (f_2 \Rightarrow f_3)$, $g_4 = f_1 \vee f_2$, $g_5 = f_1 \vee (f_2 \wedge f_3)$ and $g_6 = f_2 \wedge f_3$.

least as instantiated as the inferred mode, then the call will not violate an instantiation requirement. Modes are expressed as grounding dependencies [1].

The implementation follows the framework defined in section 5 very closely. The analyser was straightforward to implement as it is essentially two bottom-up fixpoint computations: one for $\mathcal{F}$ and the other for $\mathcal{D}$. The only subtlety is in handling the builtins. For each builtin, it is necessary to select a grounding dependency that is sufficient for avoiding an instantiation error. This is an lower approximation (the *required mode* of table 1). It is also necessary to specify behaviour on success. This is an upper-approximation (the *success mode* of table 1). The lower approximations are the assertions that are added to Prolog program to obtain a constraint logic program with assertions.

Interestingly, the success mode does not always entail the required mode. Univ (=..) illustrates this. A sufficient but not necessary condition for univ not to error is that either the first or second argument is ground. This cannot be weakened in *Pos* (but could be weakened in a type dependency domain [7] that expressed rigid lists). The success mode is that the first argument is ground iff the second argument is ground (which does not entail the required mode). Note too that keysort and sort error if their first argument is free. A sufficient mode for expressing this requirement is that the first argument is ground. Again, this requirement cannot be weakened in *Pos*.

The analyser has been applied to some standard Prolog benchmarks which can be found at http://www.oakland.edu/~l2lu/benchmarks-BG.zip. The results of the analysis, that is, the calling modes for the predicates in the smaller benchmarks, are given in table 2. The results, though surprising in some cases (see sort of permSort and insert of treesort for example) have been verified by hand and appear to be optimal for Pos. The analysis, of course, can be applied to larger programs (though it becomes very difficult to verify the results by hand) and table 3 demonstrates that the analysis scales smoothly to medium-scale programs at least. The table lists the larger benchmarks (which possibly include some unreachable code) in terms of increasing *size* measured by the total number of atoms in the source. The *abs* column records the time in milliseconds required to read, parse and normalise the source into the ground program representation used by the analyser; *lfp* is the time needed to compute the fixpoint characterising the success modes; *gfp* is the time needed to compute the calling modes; and finally *sum* is the total analysis time. This includes the (usually negligible) overhead of annotating the source with the modes required by builtins. Timings were performed on a Dell GX200 1GHz PC with 128 MB memory running Windows 2000. The timings suggest that the analysis is practical at least for medium-scale programs (though the running time for BDDs can be sensitive to the particular dependencies that arise). Moreover, with a state-of-the-art GER factorised BDD package [2] the analysis would be faster. Interestingly, the time to compute the lfp often dominates the whole analysis. BDD widening will be required to analyse very large applications but this is a study within itself [23].

7 Related work

Our work was motivated by the recent revival of interest in logic programming with assertions [6, 32]. For example, [33] argues that it is useful to trap an unexpected call to a predicate with an assertion otherwise a program may error at a point that is far from the source of the problem. Moreover, [32] 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. Our work shows how `entry` assertions can be automatically synthesised which ensure that instantiation errors do not occur while executing the program.

The most closely related work concerns the demand analysis of ccp [11, 15]. A demand analysis for the ccp language Janus [34] is proposed in [11] which determines whether or not a predicate is uni-modal. A predicate is uni-modal iff the argument tuple for each clause share 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 [12] presents a goal-dependent (forward) analysis that detects those ccp predicates which can be scheduled left-to-right without deadlock. If assertions are used to approximate synchronisation, then the analysis described in this paper can be re-interpreted as a backward suspension analysis of ccp under left-to-right scheduling.

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 [18] 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 the backward analysis presented in this paper since abduction is the inverse image of a forward semantics whereas pseudo-complement is the inverse image of conjunction – the basic computational step in forward (and backward) semantics.

The termination inference engine of [17] decomposes the cTI analyser of [29] into two components: a termination checker [8] and the backward analysis described in this paper. First, the

benchmark	*predicate*	*mode*
bubblesort	sort(x_1, x_2)	x_1
	ordered(x_1)	x_1
	append(x_1, x_2, x_3)	*true*
dnf	go	*true*
	dnf(x_1, x_2)	*true*
	norm(x_1, x_2)	*true*
	literal(x_1)	*true*
heapify	greater(x_1, x_2)	$x_1 \wedge x_2$
	adjust(x_1, x_2, x_3, x_4)	$\begin{pmatrix} (x_1 \wedge x_4) & \vee \\ (x_1 \wedge x_2 \wedge x_3) & \vee \\ (\neg x_2 \wedge \neg x_3 \wedge x_4) & \end{pmatrix}$
	heapify(x_1, x_2)	x_1
permSort	select(x_1, x_2, x_3)	*true*
	ordered(x_1)	x_1
	permutation(x_1, x_2)	*true*
	sort(x_1, x_2)	$x_1 \vee x_2$
queens	noattack(x_1, x_2, x_3)	$x_1 \wedge x_2 \wedge x_3$
	safe(x_1)	x_1
	delete(x_1, x_2, x_3)	*true*
	perm(x_1, x_2)	*true*
	queens(x_1, x_2)	$x_1 \vee x_2$
quicksort	append(x_1, x_2, x_3)	*true*
	qsort(x_1, x_2)	x_1
	partition(x_1, x_2, x_3, x_4)	$x_2 \wedge (x_1 \vee (x_3 \wedge x_4))$
treeorder	member(x_1, x_2)	*true*
	select(x_1, x_2, x_3)	*true*
	split(x_1, x_2, x_3, x_4)	*true*
	split$(x_1, \ldots, x_7)$	*true*
	visits2tree(x_1, x_2, x_3)	*true*
	v2t(x_1, x_2, x_3)	*true*
treesort	tree_to_list_aux(x_1, x_2, x_3)	*true*
	tree_to_list(x_1, x_2)	*true*
	list_to_tree(x_1, x_2)	x_1
	insert_list(x_1, x_2, x_3)	$x_1 \wedge x_2$
	insert(x_1, x_2, x_3)	$x_1 \wedge (x_2 \vee x_3)$
	treesort(x_1, x_2)	x_1

Table 2: Precision of the Mode Analysis (small benchmarks)

file	*size*	*abs*	*lfp*	*gfp*	*sum*	*file*	*size*	*abs*	*lfp*	*gfp*	*sum*
astar	100	10	10	0	20	tictactoe	258	20	10	10	40
fft	104	20	0	10	30	jons2	261	20	10	0	30
knight	105	10	0	0	10	kalah	269	30	10	20	60
browse_wamcc	106	10	0	0	10	draw	289	70	91	40	201
cal_wamcc	108	10	10	0	20	cs_r	311	40	20	10	70
life	110	10	10	10	30	reducer	320	40	30	0	70
crypt_wamcc	113	10	0	0	10	sdda	336	20	21	0	41
cry_mult	118	10	10	10	30	bryant	349	30	120	21	171
browse	125	10	10	0	20	ga	363	50	30	20	100
bid	128	10	10	0	20	neural	378	30	10	0	40
disj_r	148	30	0	10	40	press	381	30	20	0	50
consultant	151	20	0	10	30	peep	414	50	20	10	80
ncDP	156	10	10	0	20	nbody	421	40	20	20	80
tsp	162	30	20	10	60	eliza	432	50	20	0	70
elex_scanner	165	20	10	0	30	read	434	40	20	10	70
robot	165	10	10	0	20	simple_analyzer	512	90	701	20	811
sorts	172	0	10	10	20	ann	547	50	30	10	90
cs2	175	30	10	10	50	diffsimpsv	681	61	100	0	161
scc	175	10	141	0	151	arch1	692	50	40	10	100
bp0-6	201	20	10	0	30	asm	800	60	40	30	130
bnet	205	20	20	0	40	poker	962	81	70	10	161
jons	222	40	0	10	50	pentomino	981	50	40	80	170
mathlib	226	10	10	0	20	chat	1037	411	1422	1082	2915
intervals	230	20	10	10	40	sim_v5-2	1308	80	70	0	150
barnes_hut	240	40	30	40	110	semigroup	2328	180	90	60	350

Table 3: Speed of the Mode Analysis (medium-scale benchmarks)

termination inference engine computes a set of binary clauses which describe possible loops in the program with size relations. Second, a Boolean function is inferred for each predicate that describes moding conditions sufficient for each loop to only be executed a finite number of times. Third, the backward analysis described in this paper is applied to infer initial modes by calculating a greatest fixpoint which guarantee that the moding conditions hold and thereby assure termination. Interestingly, the cTI analyser involves a μ-calculus solver to compute the greatest fixpoint of an equivalent (though more complex) system of equations. This seems to suggest that greatest fixpoints are important in backward analysis.

Cousot and Cousot [10] explain how a backward collecting semantics can be deployed to precisely characterise states that arise in finite SLD-derivations. First, they present a forward collecting semantics that records the descendant states that arise from a set of initial states. Second, they present a dual (backward) collecting semantics that records those states which occur as ascendant states of the final states. By combining both semantics, they characterise the set of descendant states of the initial states which are also ascendant states of the final states of the transition system. This use of backward analysis is primarily as a device to improve the precision of a classic goal-dependent analysis. Our work is more radical in the sense that it shows how a bottom-up analysis performed in a backward fashion, can be used to characterise initial queries. Moreover it is used for lower approximation rather than upper approximation.

Mazur, Janssens and Bruynooghe [28] present a kind of *ad hoc* backward analysis to derive reuse conditions from a goal-independent reuse analysis for Mercury [36]. The analysis propagates reuse information from a point where a structure is decomposed in a clause to the point where the clause is invoked in its parent clause. This is similar in spirit to how demand is passed from a callee to a caller in the backward analysis described in this paper. However, the reuse analysis does not propagate information right-to-left across a clause using pseudo-complement, and so one interesting topic for future work will to be relate these two analyses. Another matter for future work, will be to investigate the extent to which our backward mode analysis can be reconstructed by inverting abstract functions [24].

8 Conclusion

We have shown how abstract interpretation, and specifically a backward analysis, can infer moding properties which if satisfied by the initial query, come with the guarantee that the program and query cannot generate instantiation errors. Backward analysis has other applications in termination inference and also in inferring queries for which the builtins called from within the program behave predictably in the presence of rational trees. The analysis is composed of two bottom-up fixpoint calculations, a lfp and a gfp, both of which are straightforward to implement. The lfp characterises success patterns. The gfp, uses these success patterns to infer safe initial calling patterns. It propagates moding requirements right-to-left, against the control-flow, using the pseudo-complement operator. This operator fits with backward analysis since it enables moding requirements to be minimised (maximally weakened) in right-to-left propagation. This operator, however, requires that the computational domain be closed under Heyting completion (or equivalently condense). This requirement seems reasonable because disjunctive dependencies occur frequently in right-to-left propagation and therefore significant precision would be lost if the requirement were relaxed. Experimental evaluation has demonstrated that the analysis is practical in the sense that it can infer calling modes for medium-scaled programs. Finally, our work adds weight to the belief that condensing is an important property in the analysis of logic programs.

Acknowledgements

We thank Maurice Bruynooghe, Mike Codish, Samir Genaim, Roberto Giacobazzi, Jacob Howe, Fred Mesnard, Germán Puebla and Francesca Scozzari for helpful discussions. We would also like to thank the anonymous referees for their comments and Peter Schachte for his BDD analyser. We also thank Roberto Bagnara for the use of some of the CHINA benchmarks. This work was supported, in part, by EPSRC grant GR/MO8769.

References

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

[2] R. Bagnara and P. Schachte. Factorizing Equivalent Variable Pairs in ROBDD-Based Implementations of *Pos*. In *International Conference on Algebraic Methodology and Software Technology*, volume 1548 of *Lecture Notes in Computer Science*, pages 471–485. Springer-Verlag, 1999.

[3] R. Bagnara, E. Zaffanella, R. Gori, and P. M. Hill. Boolean Functions for Finite-Tree Dependencies. In *International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 2250 of *Lecture Notes in Artificial Intelligence*, pages 575–589. Springer-Verlag, 2001.

[4] G. Birkhoff. *Lattice Theory.* AMS Press, 1967.

[5] A. Bossi, M. Gabbrielli, G. Levi, and M. Martelli. The s-Semantics Approach: Theory and Applications. *The Journal of Logic Programming*, 19/20:149–197, 1994.

[6] 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.

[7] M. Codish and V. Lagoon. Type Dependencies for Logic Programs using ACI-unification. *Theoretical Computer Science*, 238:131–159, 2000.

[8] 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.

[9] 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.

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

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

[12] S. 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.

[13] W. Drabent and J. Małuszyński. Inductive Assertion Method for Logic Programs. *Theoretical Computer Science*, 59(1):133–155, 1988.

[14] P. Dyber. Inverse Image Analysis Generalises Strictness Analysis. *Information and Computation*, 90(2):194–216, 1991.

[15] M. Falaschi, P. Hicks, and W. Winsborough. Demand Transformation Analysis for Concurrent Constraint Programs. *The Journal of Logic Programming*, 41(3):185–215, 2000.

[16] G. Filé and F. Ranzato. Improving Abstract Interpretations by Systematic Lifting to the Powerset. In *International Logic Programming Symposium*, pages 655–669. MIT Press, 1994.

[17] S. Genaim and M. Codish. Inferring Termination Conditions for Logic Programs using Backwards Analysis. In *International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 2250 of *Lecture Notes in Artificial Intelligence*, pages 681–690. Springer-Verlag, 2001.

[18] R. Giacobazzi. Abductive Analysis of Modular Logic Programs. *Journal of Logic and Computation*, 8(4):457–484, 1998.

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

[20] R. Giacobazzi, F. Ranzato, and F. Scozzari. Building Complete Abstract Interpretations in a Linear Logic-based Setting. In *Static Analysis Symposium*, volume 1503 of *Lecture Notes in Computer Science*, pages 215–229. Springer-Verlag, 1998.

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

[22] C. Hall and D. Wise. Generating Function Versions with Rational Strictness Patterns. *Science of Computer Programming*, 12:39–74, 1989.

[23] 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.

[24] R. J. M. Hughes and J. Launchbury. Reversing Abstract Interpretations. *Science of Computer Programming*, 22:307–326, 1994.

[25] J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. *The Journal of Logic Programming*, 19/20:503–581, 1994.

[26] A. Langen. *Advanced Techniques for Approximating Variable Aliasing in Logic Programs.* PhD thesis, Computer Science Department, Los Angeles, California 90089-0782, 1991.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

[31] A. Mycroft. *Abstract Interpretation and Optimising Transformations for Applicative Programs.* PhD thesis, University of Edinburgh, 1981.

[32] G. Puebla, F. Bueno, and M. 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.

[33] G. Puebla, F. Bueno, and M. 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.

[34] V. 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.

[35] F. Scozzari. Logical Optimality of Groundness Analysis. *Theoretical Computer Science*, to appear.

[36] 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.

[37] D. van Dalen. *Logic and Structure.* Springer, 1997.

[38] 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.

A Proof appendix

Proof A.1 (Proof for proposition 3.1) Proof by induction. Let $I_0 = \emptyset$, $I'_0 = \emptyset$, $I_{k+1} = \mathcal{F}^{\mathcal{C}}_P(I_k)$ and $I'_{k+1} = \mathcal{F}^{\wp^{\Downarrow}(\mathcal{C})}_{\Downarrow(P)}(I'_k)$. To show $\Downarrow(I_k) \sqsubseteq I'_k$ since then it follows that $\Downarrow(\mathrm{lfp}(\mathcal{F}^{\mathcal{C}}_P)) = \Downarrow(\sqcup_{k\in\mathbb{N}} I_k) \sqsubseteq \sqcup_{k\in\mathbb{N}} \Downarrow(I_k) \sqsubseteq \sqcup_{k\in\mathbb{N}} I'_k = \mathrm{lfp}(\mathcal{F}^{\wp^{\Downarrow}(\mathcal{C})}_{\Downarrow(P)})$. The base case is trivial so suppose $\Downarrow(I_k) \sqsubseteq I'_k$. Let $[p(\vec{x}) \coloneq c']_{\approx} \in I_{k+1}$. Then there exists $p(\vec{x}) \coloneq c, p_1(\vec{x}_1), \ldots, p_n(\vec{x}_n) \in P$ and $\{[p_i(\vec{x}_i) \coloneq c_i]_{\approx}\}_{i=1}^n \subseteq I_k$ such that $c' = c \otimes \otimes_{i=1}^n \overline{\exists}_{\vec{x}_i}(c_i)$. Observe that $\Downarrow(c') \subseteq \Downarrow(c) \cap \cap_{i=1}^n \Downarrow(\overline{\exists}_{\vec{x}_i}(c_i)) \subseteq \Downarrow(c) \cap \cap_{i=1}^n \overline{\exists}_{\vec{x}_i}(\Downarrow(c_i))$. But by the inductive hypothesis, there exist $\{[p_i(\vec{x}_i) \coloneq c'_i]_{\approx}\}_{i=1}^n \subseteq I'_k$ such that $\Downarrow(c_i) \subseteq c'_i$. Hence $[p(\vec{x}) \coloneq c'']_{\approx} \in I'_{k+1}$ such that $\Downarrow(c') \subseteq c''$ so that $I_{k+1} \sqsubseteq I'_{k+1}$ and the result follows.

Proof A.2 (Proof for theorem 3.3) The proof tactic is analogous to that used for proposition 3.1.

Proof A.3 (Proof for corollary 3.1) Let $\mathcal{C}$ be a semi-cylindric constraint system and $\rho \in uco(\wp^{\Downarrow}(\mathcal{C}))$ be a semi-morphism. By proposition 3.1 it follows that $\Downarrow(\mathcal{F}^{\mathcal{C}}(P)) \sqsubseteq \mathcal{F}^{\wp^{\Downarrow}(\mathcal{C})}(\Downarrow(P))$ and hence $\rho(\Downarrow(\mathcal{F}^{\mathcal{C}}(P))) \sqsubseteq \rho(\mathcal{F}^{\wp^{\Downarrow}(\mathcal{C})}(\Downarrow(P)))$ and by theorem 3.3 $\rho(\mathcal{F}^{\wp^{\Downarrow}(\mathcal{C})}(\Downarrow(P))) \sqsubseteq \mathcal{F}^{\mathrm{cod}(\rho)}(\rho(\Downarrow(P)))$ and so the result follows.

Proof A.4 (Proof for proposition 5.1) Let $D_{n+1} \sqsubseteq D_n$ for all $n \in \mathbb{N}$. Put $E_n = \cup\{D_l \mid l \geq n\}$ and $E = \cap\{E_n \mid n \in \mathbb{N}\}$. Since $D_{n+1} \sqsubseteq D_n$ observe that $E_n \equiv D_n$ for all $n \in \mathbb{N}$ and hence $\mathcal{D}^{\rho,\mathcal{C}}_P(\sqcap\{[D_n]_{\equiv} \mid n \in \mathbb{N}\}) = \mathcal{D}^{\rho,\mathcal{C}}_P(\sqcap\{[E_n]_{\equiv} \mid n \in \mathbb{N}\}) = \mathcal{D}^{\rho,\mathcal{C}}_P([E]_{\equiv}) = [\mathcal{D}^{\rho,\mathcal{C}}_P(E)]_{\equiv} = [\cap\{\mathcal{D}^{\rho,\mathcal{C}}_P(E_n) \mid n \in \mathbb{N}\}]_{\equiv} = \sqcap\{[\mathcal{D}^{\rho,\mathcal{C}}_P(E_n)]_{\equiv} \mid n \in \mathbb{N}\} = \sqcap\{\mathcal{D}^{\rho,\mathcal{C}}_P([E_n]_{\equiv}) \mid n \in \mathbb{N}\} = \sqcap\{\mathcal{D}^{\rho,\mathcal{C}}_P([D_n]_{\equiv}) \mid n \in \mathbb{N}\}$.

Proof A.5 (Proof for lemma 5.1) Proof by (double) induction. Let $\langle p(\vec{y}); c''; 1\rangle = s_1 \Rrightarrow^{\star}_P s_n \Rrightarrow_P \lozenge$, $s_i = \langle g_i; c_i; h_i\rangle$ and suppose $(\mathcal{D}^{\rho,\mathcal{C}}_P)^k(\top) = [D_k]_{\equiv}$. The outer induction is on k.

base case: Suppose $\max(\{\max(h_i) \mid 1 \leq i \leq n\}) \leq 1$ and $[p(\vec{y}) \coloneq e]_{\approx} \in D_1$. Thus $s_1 \Rrightarrow_P \lozenge$ so that $s_1 \Rightarrow_P \lozenge$ and hence there exists $p(\vec{x}') \coloneq C' \diamond c', g' \in P$ such that $\partial^{\vec{x}'}_{\vec{y}}(\overline{\exists}_{\vec{y}}(c'')) \notin C'$. Then $[p(\vec{x}') \coloneq e']_{\approx} \in D_1$ where $\overline{\exists}_{\vec{z}}(d_{\vec{z},\vec{y}} \otimes e) = \overline{\exists}_{\vec{z}}(d_{\vec{z},\vec{x}'} \otimes e')$ and $\mathrm{var}(\vec{z}) \cap (\mathrm{var}(\vec{y}) \cup FV(e) \cup \mathrm{var}(\vec{x}') \cup FV(e')) = \emptyset$. Observe that $e' \subseteq \overline{\forall}_{\vec{x}'}(C')$ and thus $\partial^{\vec{x}'}_{\vec{y}}(\overline{\exists}_{\vec{y}}(e)) = \overline{\exists}_{\vec{x}'}(e') \subseteq \overline{\exists}_{\vec{x}'}(\overline{\forall}_{\vec{x}'}(C')) = \overline{\forall}_{\vec{x}'}(C') \subseteq C'$. Hence $\partial^{\vec{x}'}_{\vec{y}}(\overline{\exists}_{\vec{y}}(c'')) \notin \partial^{\vec{x}'}_{\vec{y}}(\overline{\exists}_{\vec{y}}(e))$ so that $\overline{\exists}_{\vec{y}}(c'') \notin \overline{\exists}_{\vec{y}}(e)$ as required.

inductive case: Suppose $k = \max(\{\max(h_i) \mid 1 \leq i \leq n\}) > 1$ and $[p(\vec{y}) \coloneq e]_{\approx} \in D_k$. Suppose, for the sake of a contradiction, that $\overline{\exists}_{\vec{y}}(c'') \in \overline{\exists}_{\vec{y}}(e)$. Since $k > 1$ there exists $w = p(\vec{x}) \coloneq C \diamond c, p_1(\vec{x}_1), \ldots, p_l(\vec{x}_l) \in P$, $\varphi \in Ren$ such that $\varphi(\mathrm{CLP}(w)) = p(\vec{x}') \coloneq c', p_1(\vec{x}'_1), \ldots, p_l(\vec{x}'_l) \ll_{s_1} \mathrm{CLP}(P)$ and $s_2 = \langle p_1(\vec{x}'_1), \ldots, p_l(\vec{x}'_l); c'_1; 2^l\rangle$ and $c'_1 = c'' \otimes d_{\vec{y},\vec{x}'} \otimes c'$. Suppose $\langle p_1(\vec{x}'_1); c'_1\rangle \Rrightarrow^{\star}_P \langle \epsilon; c'_2\rangle, \ldots, \langle p_m(\vec{x}'_m); c'_m\rangle \Rrightarrow^{\star}_P \lozenge$. Without loss of generality assume $FV(\mathrm{CLP}(w)) \cap FV(c'_i) = \emptyset$ for all $i \in [1, m]$. Let $\vec{v} = \vec{x} \cdot \vec{x}_1 \cdots \vec{x}_l$ and $\vec{v}' = \vec{x}' \cdot \vec{x}'_1 \cdots \vec{x}'_l$. Let $g'_i \in \mathcal{C}$ such that $\langle p_i(\vec{x}'_i); 1\rangle \to^{\star}_P \langle \epsilon; g'_i\rangle$ and $c'_{i+1} = c'_i \otimes g'_i$ for all $i \in [1, m)$. For all $i \in [1, m)$, put $g_i = \partial^{\vec{x}_i}_{\vec{x}'_i}(g'_i)$. Put $c_1 = \partial^{\vec{v}}_{\vec{v}'}(c'_1)$ and for all $i \in [2, m]$, put $c_i = \partial^{\vec{x}_i}_{\vec{x}'_i}(c'_i)$. Then $c_{i+1} = c_i \otimes g_i$ for all $i \in [1, m)$. Let $\mathcal{O}^{\mathcal{C}}(P) = [F]_{\equiv}$. By proposition 4.1, $[p_i(\vec{x}_i) \coloneq g_i]_{\approx} = [p_i(\vec{x}'_i) \coloneq g'_i]_{\approx} \in F$ for all $i \in [1, m)$. By theorem 3.2, $\mathcal{O}^{\mathcal{C}}(P) = \mathcal{F}^{\mathcal{C}}(P)$ and by corollary 3.1, $\rho(\Downarrow(\mathcal{F}^{\mathcal{C}}(P))) \sqsubseteq \mathcal{F}^{\mathrm{cod}(\rho)}(\rho(\Downarrow(P)))$. Thus for $i \in [1, m)$ there exists $[p_i(\vec{x}_i) \coloneq f_i]_{\approx} \in F$ such that $\rho(\Downarrow(g_i)) \subseteq f_i$. Put $f_i = 0$ for all $i \in [m, l]$ to ensure $[p_i(\vec{x}_i) \coloneq f_i]_{\approx} \in F$ for all $i \in [m, l]$. Let $[p_i(\vec{x}_i) \coloneq d_i]_{\approx} \in D_k$ for all $i \in [1, l]$. Finally put $e_{n+1} = \mathcal{C}$, $e_i = d_i \cap (f_i \to' e_{i+1})$ for all $i \in [1, l]$ and $e_0 = C \cap (\rho(\Downarrow(c)) \to' e_1)$. The inner induction is on i and is used to show $\rho(\Downarrow(c_i)) \subseteq e_i$ for all $i \in [1, m]$.

base case: Now $c_1 = \partial^{\vec{v}}_{\vec{v}'}(c'_1) \trianglelefteq \partial^{\vec{x}}_{\vec{y}}(\overline{\exists}_{\vec{y}}(c'')) \in C$. Thus $\rho(\Downarrow(c_1)) \subseteq C$. Furthermore, $c_1 = \partial^{\vec{v}}_{\vec{v}'}(c'_1) \trianglelefteq \partial^{\vec{v}}_{\vec{v}'}(c') = c$. Thus $\rho(\Downarrow(c_1)) \subseteq \rho(\Downarrow(c))$. Moreover, $c_1 \trianglelefteq \partial^{\vec{x}}_{\vec{y}}(\overline{\exists}_{\vec{y}}(c'')) \in \partial^{\vec{x}}_{\vec{y}}(\overline{\exists}_{\vec{y}}(e)) \subseteq \overline{\forall}_{\vec{x}}(e_0) \subseteq e_0$. Thus $\rho(\Downarrow(c_1)) \subseteq e_0$. However, $e_0 = C \cap (\rho(\Downarrow(c)) \rightarrow' e_1)$. Thus $\rho(\Downarrow(c_1)) \subseteq C \cap (\rho(\Downarrow(c)) \rightarrow' e_1)$ and $\rho(\Downarrow(c_1)) \subseteq \rho(\Downarrow(c_1)) \cap C \cap (\rho(\Downarrow(c)) \rightarrow' e_1) = \rho(\Downarrow(c_1)) \cap (\rho(\Downarrow(c)) \rightarrow' e_1)$ $= \rho(\Downarrow(c_1)) \cap \rho(\Downarrow(c)) \cap (\rho(\Downarrow(c)) \rightarrow' e_1) = \rho(\Downarrow(c_1)) \cap \rho(\Downarrow(c)) \cap e_1 = \rho(\Downarrow(c_1)) \cap e_1$. Therefore $\rho(\Downarrow(c_1)) \subseteq e_1$ as required.

inductive case: Suppose $\rho(\Downarrow(c_i)) \subseteq \varphi(e_i)$. Now $\rho(\Downarrow(c_{i+1})) = \rho(\Downarrow(c_i \otimes g_i)) \subseteq \rho(\Downarrow(c_i)) \cap \rho(\Downarrow(g_i)) \subseteq e_i \cap \rho(\Downarrow(g_i)) \subseteq e_i \cap \rho(\Downarrow(g_i)) \subseteq e_i \cap f_i \subseteq (f_i \rightarrow' e_{i+1}) \cap f_i = e_{i+1}$. Therefore $\rho(\Downarrow(c_{i+1})) \subseteq e_{i+1}$ as required.

Thus $\rho(\Downarrow(c_m)) \subseteq e_m \subseteq d_m$ so that $c_m \in d_m$. Let $d'_m = \partial^{\vec{x}'_m}_{\vec{x}_m}(\overline{\exists}_{\vec{x}_m}(d_m))$ and observe that $[p_m(\vec{x}'_m) \text{ :- } d'_m]_\approx = [p_m(\vec{x}_m) \text{ :- } d_m]_\approx \in D_k$. Put $c''_m = \partial^{\vec{x}'_m}_{\vec{x}_m}(\overline{\exists}_{\vec{x}_m}(c_m))$ so that $c'_m \trianglelefteq c''_m \in d'_m$. By the inductive hypothesis $\langle p_m(\vec{x}'_m); c'_m\rangle \not\Rightarrow^\star_P \Diamond$ which is a contradiction and hence $\overline{\exists}_{\vec{y}}(c'') \notin \overline{\exists}_{\vec{y}}(e)$ as required.

The result follows.

Proof A.6 (Proof for theorem 5.1) Let $\mathcal{D}^{\rho,\mathcal{C}}(P) = [D]_\equiv$, $[p(\vec{y}) \text{ :- } e]_\approx \in D$ and $c'' \in \overline{\exists}_{\vec{y}}(e)$. Thus $\overline{\exists}_{\vec{y}}(c'') \in \overline{\exists}_{\vec{y}}(\overline{\exists}_{\vec{y}}(e)) = \overline{\exists}_{\vec{y}}(e)$. Suppose, for the sake of a contradiction, that $\langle p(\vec{y}); c''; 1\rangle = s_1 \Rightarrow^\star_P s_n \Rightarrow_P \Diamond$ where $s_i = \langle g_i; c_i; h_i\rangle$. Let $k = max(\{\max(h_i) \mid 1 \leq i \leq n\})$. Suppose $(\mathcal{D}^{\rho,\mathcal{C}}_P)^k(\top) = [D_k]_\equiv$. Since $D \sqsubseteq D_k$ and by lemma 5.1 there exists $[p(\vec{y}) \text{ :- } e']_\approx \in D_k$ such that $\overline{\exists}_{\vec{y}}(c'') \notin \overline{\exists}_{\vec{y}}(e')$. Since $\overline{\exists}_{\vec{y}}(e) \subseteq \overline{\exists}_{\vec{y}}(e')$ it follows that $\overline{\exists}_{\vec{y}}(c'') \notin \overline{\exists}_{\vec{y}}(e)$ which is a contradiction. The result follows.