Analysis of Logic Programs with Delay

Andrew Heaton[1] and Pat Hill[1] and Andy King[2]

[1] School of Computer Studies, University of Leeds, LS2 9JT, UK
[2] Computing Laboratory, University of Kent at Canterbury, CT2 7NF, UK

Abstract. The paper focuses on practical analyses for logic programs with delay. The method described is for downward-closed program properties and, in particular, groundness. A program transformation is defined which eliminates the delay statements but still enables an accurate approximation of the behaviour of delayed goals to be traced by the analyser. An implementation has been built which shows that the analysis can be both accurate and efficient.

1 Introduction

Second-generation logic programming languages, such as Gödel, IF/Prolog, SICStus Prolog provide flexible computation rules in which goals delay if their arguments are insufficiently instantiated. Goals are reawoken, later on, if their arguments become further instantiated. In these languages the default computation rule is left to right. Flexible computation rules can provide a sound treatment of negation, underpin constrained search, improve termination behaviour of programs and allow co-routining. The program permute illustrates the use of the block declaration of SICStus Prolog.

```
permute([], []).
permute([u|x1], y) :-
    remove(u, y, z), permute(x1, z).

remove(x, [u|u1], z) :-
    new_remove(u, u1, x, z).

:- block new_remove(?, ?, -, -).
new_remove(x, x, z, z).
new_remove(x, u, y1, [u|z1]) :-
    remove(x, y1, z1).

ordered([]).
ordered([x]).
ordered([x, y|z]) :-
    new_ordered(x, y, z).

:- block new_ordered(-, ?, ?), new_ordered(?, -, ?).
new_ordered(x, y, ys) :-
    x ≤ y, ordered([y|ys]).
```

Declaratively, the predicate permute/2 is true iff the first argument is a permutation of the second. remove/3 holds iff the second and third arguments are lists and the third can be obtained from the second by removing an element which corresponds to the first argument. The block declarations ensure that permute terminates for all queries. In the declaration :- block new_remove$(?, ?, -, -)$, the "-" in the third and fourth argument positions means that in a call to new_remove$/4$, if the third and fourth arguments are both uninstantiated then the call will delay until one of these arguments becomes instantiated. Without the new_remove block declaration, the query ?- permute$([1,2],x)$. would backtrack into an infinite loop after producing the single solution $x=[1,2]$[3].

The query ?-ordered(y), permute$([2,1],y)$. is an example of the generate-and-test paradigm and illustrates how block declarations can be used to improve efficiency. The block declaration for new_ordered$/3$ delays goals until *both* the first and second arguments are non-variable. This causes the ordered$/1$ goals to co-routine (interleave) with permute$/2$ goals constraining the search and reducing backtracking.

Groundness analysis detects which program variables are bound to ground terms and is important in detecting determinacy, simplifying unification, *etc*. In [19] it is also shown that groundness information can be used to simplify delay conditions, thus enabling transformations which give significant improvements in performance. In particular this allows the elimination of redundant delay declarations. Consider a groundness analysis for the query ?- ⓐpermute$([1,2],y)$ⓑ. where ⓐ and ⓑ denote program points. Groundness analysis should infer that y is ground at ⓑ but may be either ground or non-ground at ⓐ. Frameworks capable of tracing groundness in the context of co-routining, however, tend to be either imprecise or inefficient. This is reviewed in the related work section at the end of the paper. One of the major problems is in tracing the behaviour of delayed goals. Our method deals with this problem by transforming programs into abstract programs which have no delay statements but which can still trace the effects of the delaying goals. This enables potentially very efficient analyses.

In [5] a highly efficient groundness analysis using the Pos domain is given based on abstract compilation techniques. Following these techniques, we show that the Pos domain gives an accurate analysis for the special case of groundness. For example, with the above query to permute, our method will detect that y will be ground at program point ⓑ. The method is simple to implement, and the results for the programs tested so far indicate an efficient analysis for small and medium-sized programs.

The rest of the paper is organised as follows. In the next section a worked example demonstrating our approach is given. Section 3 gives the standard definitions and notation used. Section 4 defines the Pos domain whilst section 5 formalizes the program transformation central to the analysis and describes the abstract semantics. Section 6 outlines the implementation method whilst sections 7 and 8 summarise related and future work respectively.

2 Worked Example

To illustrate our approach to groundness analysis, consider the following query program below with initial goal :- query(x,y) annotated with program points ⓐ, ⓑ, ⓒ, ⓓ

[3] Note that an additional block permute$(-,-)$ declaration might also be useful since the query ?- permute(x,y) enumerates its solutions in the following order: $x=[\ ], y=[\ ]; x=[z_1], y=[z_1]; x=[z_1,z_2], y=[z_1,z_2]; x=[z_1,z_2,z_3], y=[z_1,z_2,z_3]$ *etc*, missing solutions such as $x=[z_1,z_2], y=[z_2,z_1]$. Unfairness can be avoided by delaying the call permute(x,y) until either of its arguments are instantiated.

and ⓔ.

$$\begin{array}{l}\mathsf{query}(x,y) \text{ :- } ⓐ\mathsf{delay}(x,y), ⓑ x = [\,]\ ⓒ\ .\\ \text{:- block } \mathsf{delay}(-,?).\\ \mathsf{delay}(x,y) \text{ :- } ⓓ x = y\ ⓔ\ .\end{array}$$

The equations $\phi_a = \phi_b = \varepsilon$, $\phi_c = \{x = [\,], y = [\,]\}$ represent the actual bindings on x and y at ⓐ, ⓑ and ⓒ respectively.

Our method basically transforms the program to a Datalog program which, although free from delay statements, can still trace synchronisation behaviour. The abstract program for query is listed below. Let $gr(x)$ denote that x is a ground term. The unification $x = [\,]$ has been abstracted by the propositional formula $gr(x)$. The block declaration is modelled by the formula $gr(x) \rightarrow ((gr(x) \leftrightarrow gr(x')) \wedge (gr(y) \leftrightarrow gr(y')))$. The unification $x = y$ has been abstracted by the propositional formula $gr(x') \leftrightarrow gr(y')$. The intention is that x' and y' will be unified with x and y whenever x is a ground term.

$$\begin{array}{ll}\mathsf{query}(x,y) \text{ :- } & ⓐ\mathsf{delay}(x,y), ⓑ\{gr(x)\}\ ⓒ\\ \mathsf{delay}(x,y) \text{ :- } & ⓓ\{gr(x) \rightarrow ((gr(x) \leftrightarrow gr(x')) \wedge (gr(y) \leftrightarrow gr(y')))\\ & \quad \wedge(gr(x') \leftrightarrow gr(y'))\}\ ⓔ\end{array}$$

Restricting the formula on the right hand side of $\mathsf{delay}(x,y)$ to just the variables x, y we get the formula $gr(x) \rightarrow (gr(x) \leftrightarrow gr(y))$. At the program points ⓓ and ⓔ, $gr(x)$ holds and so at ⓔ the abstract description will be $gr(x) \wedge (gr(x) \leftrightarrow gr(y))$. This is logically equivalent to $gr(x) \wedge gr(y)$. Hence it has been deduced that x and y are both ground at ⓔ and so this is the answer pattern for delay.

With the block statements removed, the call and answer patterns can now be computed automatically by transforming the abstract program with a query-answer transformation and then calculating the minimal model. Implementation is thus straightforward even though synchronisation behaviour is accurately traced. Note that, in the case of the permute program, the analysis is powerful enough to infer that if the call pattern for permute(x,y) has x ground, then the answer pattern will have both x and y ground.

3 Preliminaries

3.1 Syntax of logic programs

Var denotes the set of variables, $Term$ the set of terms, $Pred$ the set of predicate symbols, $Atom$ the set of atoms of the form $p(\mathbf{x})$ where p is a predicate symbol and $\mathbf{x}$ are distinct variables. Eqn denotes the set of finite sets of equations of the form $a = b$ where either $a, b \in Term$ or $a, b \in Atom$. A literal is either an atom or an equation whereas a goal is a finite sequence of literals. The sets of literals and goals are denoted by Lit and $Goal$ respectively. A clause is a syntactic object of the form $h \leftarrow \mathbf{b}$ where h, the head, is an atom and $\mathbf{b}$, the body, is a finite sequence of literals. $Prog$ is the set of programs, that is, the set of finite sets of clauses. Var, $Term$, $Atom$, Eqn, Lit, $Goal$ and $Prog$ respectively have typical members v, w, x, y, z; t; a, b; e; l; G and P. P_G denotes a program P with initial goal G. Initial goals are restricted to be single atoms. Also $\mathrm{var(obj)}$ denotes the set of variables in a syntactic object obj, :: denotes concatenation of sequences, and ϵ denotes the empty sequence.

The set of idempotent substitutions from Var to $Term$ is denoted Sub and the set of renamings (which are bijective mappings from Var to Var) is denoted Ren. Sub and Ren extend in the usual way from functions from variables to terms, to functions from terms to terms, to functions from substitutions to substitutions, to functions from atoms to atoms and to functions from clauses to clauses. The restriction of a substitution θ to a set of variables U is denoted by $\theta \upharpoonright U$ and the composition of two substitutions θ and ϑ is denoted by $\theta \circ \vartheta$ and defined such that $(\theta \circ \vartheta)(u) = \theta(\vartheta(u))$. ε denotes the empty substitution and sets of substitutions will usually be denoted by Φ and Ψ. There is a natural mapping from substitutions to equations, that is, $eqn(\theta) = \{u = t \mid u \mapsto t \in \theta\}$, and $\mathrm{mgu(E)}$ denotes the set of most general unifiers for an equation set E.

Let $\leq$ denote the instance ordering, that is, $\theta \leq \theta'$ iff there exists a substitution $\sigma \in Sub$ such that $\theta = \sigma \circ \theta'$. Similarly, a syntactic object o is an instance of another o', denoted $o \leq o'$, iff there exists a substitution σ such that $o = \sigma(o')$. Instance lifts to $\wp(Atom)$ by $I \leq I'$ iff for all $a \in I$ there exists $a' \in I'$ such that $a \leq a'$. This, in turn, defines the equivalence relation $\sim$, that is $I \sim I'$ iff $I \leq I'$ and $I' \leq I$. Finally, the (quotiented) set of interpretations $Int = \wp(Atom)/\sim$ is ordered by $[I]_\sim \leq [I']_\sim$ iff $I \leq I'$.

3.2 Operational semantics of logic programs with delay

The operational semantics is described in terms of reductions between states. The set of states is defined by $State = Goal \times Sub \times Goal$. A state $\langle G, \theta, D\rangle$ records a sequence of literals G, the current substitution θ, and a sequence of delayed atoms D. For a given state s and a program P, the relation $\ll_s$ is defined such that $h \leftarrow \mathbf{b} \ll_s P$ iff there exists $\rho \in Ren$ such that $\rho(h \leftarrow \mathbf{b}) \in P$ and $\mathrm{var(h} \leftarrow \mathbf{b}) \cap \mathrm{var(s)} = \emptyset$. Similar to [17], to abstract away from particular language considerations, the operational semantics is defined in terms of two parametric functions $reduce$ and $woken$. (In [17], a $delay$ truth function is used instead with $delay \equiv \neg reduce$.) We say the atom a reduces with the substitution θ whenever $reduce(a, \theta)$ holds, whereas $woken(D, \theta)$ denotes the subset of atoms in D (ordered as a sequence) that are woken with the substitution θ. These functions are assumed to satisfy four conditions:

1. $a \in woken(D, \theta)$ iff $a \in D$ and $reduce(a, \theta)$.
2. $reduce(a, \theta)$ iff $reduce(\rho(a), \rho(\theta))$ for all $\rho \in Ren$.
3. $reduce(a, \theta)$ iff $reduce(a, \theta \upharpoonright \mathrm{var(a)})$.
4. if $\theta \leq \vartheta$ and $reduce(a, \vartheta)$ then $reduce(a, \theta)$.

Definition 1 reduction. The relation $s \rightarrow_P s'$ where $s, s' \in State$ is defined as follows:

- if $s = \langle x = t :: G, \theta, D\rangle$ then $s' = \langle D' :: G, \theta', D \setminus D'\rangle$ where $\theta' \in \mathrm{mgu}(\{\mathrm{x = t}\} \cup \mathrm{eqn}(\theta))$ and $D' = woken(D, \theta')$;
- if $s = \langle a :: G, \theta, D\rangle$ and $reduce(a, \theta)$ holds then $s' = \langle \mathbf{b} :: G, \theta', D\rangle$ where $h \leftarrow \mathbf{b} \ll_s P$ and $\theta' \in \mathrm{mgu}(\{\mathrm{h = a}\} \cup \mathrm{eqn}(\theta))$;
- if $s = \langle a :: G, \theta, D\rangle$ and $reduce(a, \theta)$ fails then $s' = \langle G, \theta, a :: D\rangle$.

Note that when an atom a has no defining clause, $reduce(a, \theta)$ fails and it is added to the delay sequence. In practise such an atom will usually cause an error. These atoms are reduced, however, to avoid the need for an error state and thus provide a simple basis for analysis. Also the relative execution order of simultaneously reawoken goals is left undefined to reflect the scheduling behaviour of implementations like SICStus [3] and IF/Prolog [20]. Our analysis defined below is not dependent on any particular scheduling order for these atoms.

Definition 2 derivation. A derivation from a state s for a program P is a (finite or infinite) sequence of reductions $s_1 \rightarrow_P s_2 \rightarrow_P \ldots$ where $s = s_1$.

The operational semantics of a program is in terms of its qualified answers. Given a derivation from a state s and a program P with last state $\langle \epsilon, \theta, D\rangle$ we say the tuple $\langle \theta, D\rangle$ is a qualified answer to s. It is successful if $D = \epsilon$ and it flounders otherwise. Call patterns can now be defined in terms of the operational semantics.

Definition 3 call patterns. For a given program P and initial goal G,

$$\mathrm{call(P_G)} = [\{\langle \mathrm{l}, \theta \restriction \mathrm{var(l)}\rangle \mid \langle \mathrm{G}, \varepsilon, \epsilon\rangle \rightarrow_{\mathrm{P}}{}^{*} \langle \mathrm{l :: G'}, \theta, \mathrm{D}\rangle\}]_{\sim}$$

3.3 Galois connections

A Galois connection between $D^{\mathcal{A}}$ and D is a 4-tuple $\langle D\ (\sqsubseteq), \alpha, D^{\mathcal{A}}\ (\sqsubseteq^{\mathcal{A}}), \gamma\rangle$ where $D\ (\sqsubseteq)$ and $D^{\mathcal{A}}\ (\sqsubseteq^{\mathcal{A}})$ are posets; $\alpha : D \to D^{\mathcal{A}}$ and $\gamma : D^{\mathcal{A}} \to D$; α and γ are monotonic; $d \sqsubseteq \gamma(\alpha(d))$ for all $d \in D$ and $\alpha(\gamma(d^{\mathcal{A}})) \sqsubseteq^{\mathcal{A}} d^{\mathcal{A}}$ for all $d^{\mathcal{A}} \in D^{\mathcal{A}}$.

We next formalise the notion of a downward-closed property, following a similar approach to [6].

Definition 4 downward-closure.

- Given a poset $D\ (\sqsubseteq)$, the downward-closure of $S \subseteq D$ is defined by $down(S) = \{d \in D \mid \exists s \in S.d \sqsubseteq s\}$. S is downward-closed iff $down(S) = S$;
- A Galois connection $\langle D\ (\sqsubseteq), \alpha, D^{\mathcal{A}}\ (\sqsubseteq^{\mathcal{A}}), \gamma\rangle$ is downward-closed iff for all $d^{\mathcal{A}} \in D^{\mathcal{A}}$ $\gamma(d^{\mathcal{A}})$ is downward-closed.

A predicate p on a poset $D\ (\sqsubseteq)$ is a subset of D and therefore is downward-closed iff the set p is downward-closed. A truth-function is downward-closed iff the predicate it defines is downward-closed.

Example 1. Given the standard instance ordering on terms, the unary predicates gr and nv are downward-closed where gr and nv are defined such that $gr(t)$ holds iff t is ground, and $nv(t)$ holds iff t is non-variable. □

4 Abstract Domains

Abstract substitutions A Pos domain is used to capture positive information about downward-closed predicates. Following [8], we let $\Omega_S(\{\wedge, \vee, \leftrightarrow, \neg\})$ denote the set of propositional formulae formed from the set of connectives $\{\wedge, \vee, \leftrightarrow, \neg\}$ and a set of propositional symbols S. A truth assignment is a function $r : S \to \{0, 1\}$. Given $f, f' \in \Omega_S(\{\wedge, \vee, \leftrightarrow, \neg\})$, $r \models f$ denotes that r satisfies f and the notation $f \models f'$ abbreviates $r \models f$ implies $r \models f'$. Two formulae are logically equivalent, $f \equiv f'$, iff $f \models f'$ and $f' \models f$.

To enable an analysis to simultaneously trace and relate multiple downward-closed properties, like groundness and non-variable (denoted from here on by gr and nv respectively), on a set of program variables V, Pos is defined in terms of a set of propositional symbols $\{gr(v), nv(v) | v \in V\}$. More generally, given n properties, $p_1, \ldots, p_n$, the set of propositional symbols is $\{p_i(v) \mid v \in V \wedge 1 \leq i \leq n\}$. As notation, we will usually add the $\sharp$ annotation to denote abstract substitutions, that is, to indicate propositional formulae.

Definition 5. $Pos_V = \{f \in \Omega_S(\{\wedge, \vee, \leftrightarrow, \neg\}) \mid u \models f\}$ where $S = \{p_i(v) | v \in V\}$ and $u : S \rightarrow \{1\}$ denotes the unit truth assignment.

Note that if Pos_V is defined for a single predicate p_1 then the definition can be simplified so that just the variables themselves represent the property. This special case coincides with the groundness analysis of [5, 8, 18]. $Pos_V / \equiv (\models)$ is a complete lattice with lub $\vee$ and glb $\wedge$ and the least and greatest elements $false$ and $true$ [8].

Abstraction and concretisation of substitutions To formalise the relationship between a substitution and a formula an auxiliary function $assign_V$ is introduced. Because $p_i(v)$ can be used both as a propositional symbol and to assert that p_i holds for v, for clarity, in what follows a truth-function f_i is used to test for a property in the concrete whereas p_i is used for a symbol in abstract formulae.

Definition 6. $assign_V(\phi) = \bigwedge_{i=1}^{n} \bigwedge_{v \in V} q_i(v)$ where $q_i(v) = \begin{cases} p_i(v) & \text{if } f_i(v) \\ \neg p_i(v) & \text{if } \neg f_i(v) \end{cases}$

Note that $assign_V(\phi)$ is not necessarily positive.

Example 2. Suppose $V = \{x, y, z\}$, $p_1 = gr$, $p_2 = nv$, $\phi = \{x \mapsto f(y)\}$ and $\phi' = \{x \mapsto f(1), y \mapsto 1\}$ (note that $\phi' \leq \phi$). Then
$assign_V(\phi) = \neg gr(x) \wedge nv(x) \wedge \neg gr(y) \wedge \neg nv(y) \wedge \neg gr(z) \wedge \neg nv(z)$ and
$assign_V(\phi') = gr(x) \wedge nv(x) \wedge gr(y) \wedge nv(y) \wedge \neg gr(z) \wedge \neg nv(z)$. □

We can now define the concretisation and abstraction functions γ and α.

Definition 7.

$$\begin{array}{ll} \gamma_V : Pos_V \rightarrow \wp(Sub), & \alpha_V : \wp(Sub) \rightarrow Pos_V \\ \gamma_V(\phi^\sharp) = \{\phi \mid \forall \varphi \leq \phi . assign_V(\varphi) \models \phi^\sharp\}, & \alpha_V(\Phi) = \bigwedge_{\Phi \subseteq \gamma_V(\phi^\sharp)} \phi^\sharp \end{array}$$

Note that defining γ_V on Pos_V ensures that $\alpha_V(\Phi) \in Pos_V$.

Example 3. Suppose $V = \{x, y, z\}$, $p_1 = gr$, $p_2 = nv$ and $\phi = \{x \mapsto f(y)\}$. Let $\phi^\sharp = (gr(x) \leftrightarrow gr(y)) \wedge nv(x)$. Then $\phi \in \gamma_V(\phi^\sharp)$ since $assign_V(\phi') \models \phi^\sharp$ for all $\phi' \leq \phi$. □

We now have a Galois connection between the abstract and concrete domains as stated below.

Proposition 8. $\langle \wp(Sub)\ (\subseteq), \alpha_V, Pos_V\ (\models), \gamma_V \rangle$ is a Galois connection.

We also want to abstract concrete interpretations, which we shall use to define abstract call patterns. This is done by pairing atoms with formulae as follows:

Definition 9.

$$Atom^\sharp = \{\langle p(\mathbf{x}), \phi^\sharp \rangle \mid p \in Pred \wedge \phi^\sharp \in Pos_{\mathrm{var}(\mathbf{x})}\}$$

The concretisation and abstraction functions can be naturally extended to interpretations as follows:

Definition 10.

$$\begin{array}{l} \gamma : \wp(Atom^\sharp) \to Int \\ \gamma(I^\sharp) = [\{\langle p(\mathbf{x}), \phi \restriction \mathrm{var}(\mathbf{x})\rangle \mid \langle \mathrm{p}(\mathbf{x}), \phi^\sharp \rangle \in \mathrm{I}^\sharp \wedge \phi \in \gamma_{\mathrm{var}(\mathbf{x})}(\phi^\sharp)\}]_\sim \end{array}$$

γ induces a natural equivalence relation on $\wp(Atom^\sharp)$, that is $I^\sharp \approx I^{\sharp'}$ iff $\gamma(I^\sharp) = \gamma(I^{\sharp'})$. This in turn is used to define $Int^\sharp = \wp(Atom^\sharp)/\approx$, the abstract analogue of Int. An ordering for $Int^\sharp$ is given by $[I^\sharp]_\approx \sqsubseteq [I^{\sharp'}]_\approx$ iff $\gamma([I^\sharp]_\approx) \subseteq \gamma([I^{\sharp'}]_\approx)$ and the lub is defined by $\sqcup_i [I^\sharp_i]_\approx = [\cup_i I^\sharp_i]_\approx$.

Definition 11.

$$\gamma : Int^\sharp \to Int \quad \alpha : Int \to Int^\sharp$$

$$\gamma(I^\sharp) = [\gamma(I^\sharp)]_\approx \; \alpha(I) = \bigsqcup_{I \subseteq \gamma([I^\sharp]_\approx)} [I^\sharp]_\approx$$

We can thus prove:

Proposition 12. $\langle Int^\sharp\ (\sqsubseteq), \alpha, Int\ (\leq), \gamma\rangle$ is a Galois connection.

5 Abstract Compilation

First we define the set of monotonic formulae, Mon_V.

Definition 13. $Mon_V = \Omega_S(\{\wedge, \vee\})$ where $S = \{p_i(v) \mid v \in V\}$.

For technical reasons apparent in the proof for safety, we define the abstract reduce function in terms of monotonic formulae. We will use certain properties of monotonic formulae in the proof. Here it suffices just to note that $Mon_V \subseteq Pos_V$. See, e.g. [8], for details.

Definition 14.

$$reduce^\sharp(p(\mathbf{x}), \phi^\sharp) \leftrightarrow (\phi^\sharp \in Mon_{\mathrm{var}(\mathbf{x})} \wedge \forall \phi \in \gamma_{\mathrm{var}(\mathbf{x})}(\phi^\sharp)\ .reduce(p(\mathbf{x}), \phi)).$$

The intention is that an abstract goal will never reduce if its corresponding concrete goal never reduces.

Example 4. Let us define $reduce^\sharp(p(\mathbf{x}), \phi^\sharp) \leftrightarrow \forall \phi \in \gamma_{\mathrm{var}(\mathbf{x})}(\phi^\sharp)\ .reduce(p(\mathbf{x}), \phi)$. In SICStus Prolog, given the declaration :- block $p(d_1, ..., d_n)$ for the n-ary predicate p, where $d_i \in \{-, ?\}$ for $1 \le i \le n$, then an abstract reduce function is:

$$reduce^\sharp(\mathrm{p}(\mathbf{x}), \phi^\sharp) \leftrightarrow \phi^\sharp \models \bigvee_{\mathrm{d_i}=-} nv(x_i)$$

□

Next we define the program transformation central to the analysis. For clarity, we introduce a notation for writing propositional formulae in terms of their variables rather than the properties of the variables (which strictly speaking are the propositional symbols), e.g. given the propositional formula $q(p_1(x), ..., p_n(x))$ we denote this by $q(x)$.

Definition 15 abstract compilation α.

$$\begin{aligned}
\alpha[\![c_1, \ldots, c_n]\!] &= \alpha_{\mathrm{clause}}[\![c_1]\!], \alpha_{\mathrm{bot}}[\![c_1]\!] \ldots, \alpha_{\mathrm{clause}}[\![c_n]\!], \alpha_{\mathrm{bot}}[\![c_n]\!] \\
\alpha_{\mathrm{clause}}[\![p(\mathbf{x}) \leftarrow \mathbf{b}]\!] &= p(\mathbf{x}') \leftarrow (\phi_r^\sharp \rightarrow \phi_s^\sharp), \alpha_{\mathrm{goal}}[\![\mathbf{b}]\!] \\
\alpha_{\mathrm{bot}}[\![p(\mathbf{x}) \leftarrow \mathbf{b}]\!] &= p(\mathbf{x}) \leftarrow \phi_b^\sharp \\
\alpha_{\mathrm{goal}}[\![l_1, \ldots, l_n]\!] &= \alpha_{\mathrm{literal}}[\![l_1]\!], \ldots, \alpha_{\mathrm{literal}}[\![l_n]\!] \\
\alpha_{\mathrm{literal}}[\![p(\mathbf{x})]\!] &= p(\mathbf{x}) \\
\alpha_{\mathrm{literal}}[\![t = t']\!] &= \phi_t^\sharp
\end{aligned}$$

where

$$\begin{cases}
\phi_b^\sharp = \bigwedge_i \bigwedge_{v \in \mathbf{x}} p_i(v) \\
\phi_r^\sharp = \bigvee_{reduce^\sharp(p(\mathbf{x}'), \phi^\sharp)} \phi^\sharp \\
\phi_s^\sharp = \alpha^\sharp_{\mathrm{var}(\mathbf{x}=\mathbf{x}')}(\mathrm{mgu}(\{\mathbf{x} = \mathbf{x}'\})) \\
\phi_t^\sharp = \alpha^\sharp_{\mathrm{var}(\mathrm{t}=\mathrm{t}')}(\mathrm{mgu}(\{\mathrm{t} = \mathrm{t}'\}))
\end{cases}$$

and $\mathbf{x}'$ are fresh variables, that is, $\mathrm{var}(\mathbf{x}') \cap \mathrm{var}(\mathrm{p}(\mathbf{x}) \leftarrow \mathbf{b}) = \emptyset$.

Example 5. The abstract program for the program permute listed in section 1 is below. First note we have written new to denote the predicate new_remove. The first (normalised) clause for permute is

permute(u, v):- $u = [\,], v = [\,]$.

This is abstracted by the propositional formula $nv(u) \wedge gr(u) \wedge nv(v) \wedge gr(v)$. When α_{bot} is applied to this same clause the propositional formula $\phi_b^\sharp$ obtained is again $nv(u) \wedge gr(u) \wedge nv(v) \wedge gr(v)$.
Next we turn to how the block declaration

:- block new_remove$(?, ?, -, -)$.

has been abstracted. Following Example 4, the abstract reduce condition is modelled as follows:

$reduce^\sharp($new_remove$(u, v, w, x), \phi^\sharp) \leftrightarrow \phi^\sharp \models nv(w) \vee nv(x)$.

For convenience, a separate predicate reduce is defined. This essentially consists of the propositional formula $(\phi_r^\sharp \rightarrow \phi_s^\sharp)$ as given in the formal definition above.

```
permute(u, v) :-
   { nv(u) ∧ gr(u) ∧ nv(v) ∧ gr(v) }.
permute(u, v) :-
   { nv(u) ∧ gr(u) ↔ (gr(w) ∧ gr(x)) },
   remove(w, v, y),
   permute(x, y).

remove(u, v, w) :-
   { nv(v) ∧ (gr(v) ↔ (gr(x) ∧ gr(y))) },
   new(u, x, y, w).
remove(u, v, w) :-
   { nv(v) ∧ nv(v) ∧ nv(x) ∧
     gr(v) ∧ gr(v) ∧ gr(x) }.

new(u, v, w, x) :-
   reduce(new(u, v, w, x), new(u', v', w', x')),
   { nv(u') ↔ nv(v') ∧ gr(u') ↔ gr(v') ∧
     nv(w') ↔ nv(x') ∧ gr(w') ↔ gr(x') }.
new(u, v, w, x) :-
   reduce(new(u, v, w, x), new(u', v', w', x')),
   { nv(x') ∧ (gr(x') ↔ (gr(v') ∧ gr(y))) },
   remove(u', w', y).
new(u, v, w, x) :-
   { nv(u) ∧ nv(v) ∧ nv(w) ∧ nv(x) ∧
     gr(u) ∧ gr(v) ∧ gr(w) ∧ gr(x) }.

reduce(new(u, v, w, x), new(u', v', w', x')) :-
   { (nv(w) ∨ nv(x)) →
       nv(u) ↔ nv(u') ∧ gr(u) ↔ gr(u') ∧
       nv(v) ↔ nv(v') ∧ gr(v) ↔ gr(v') ∧
       nv(w) ↔ nv(w') ∧ gr(w) ↔ gr(w') ∧
       nv(x) ↔ nv(x') ∧ gr(x) ↔ gr(x') }.
```

□

Note that having α_{bot} as part of the definition will lose no precision in the analysis. This is because it merely adds the possibility that all properties might hold at (almost!) any point in the program. But since only downward-closed properties are considered, this is already implicitly assumed. We are, however, making sure that all the abstract programs will terminate. If termination is not ensured then information may be lost in the case when delayed goals remain unexecuted in the concrete program.

Example 6. Suppose we had defined $\alpha[\![c_1, \ldots, c_n]\!] = \alpha_{\text{clause}}[\![c_1]\!], \ldots, \alpha_{\text{clause}}[\![c_n]\!]$. Consider the call and answer patterns of the loop program (listed below on the left) with an initial goal go, and the corresponding abstract program (listed on the right) defined by $loop^\sharp = \alpha[\![loop]\!]$ for a Pos domain that just traces groundness, that is, $p_1 = gr$.

```
go :- p(v), r(v).                go :- p(v), r(v).
:- block p(-).                   p(w) :- (gr(w) → (gr(w) ↔ gr(x))), q(x).
p(w) :- q(w).                    q(y) :- q(y).
q(y) :- q(y).                    r(z).
r(z).
```

The only derivation for loop flounders and is finite

$$\langle \mathsf{go}; \varepsilon; \epsilon\rangle \rightarrow_{loop} \langle p(w), r(w); \varepsilon; \epsilon\rangle \rightarrow_{loop} \langle r(w); \varepsilon; p(w)\rangle \rightarrow_{loop} \langle \epsilon; \{w \mapsto z\}; p(w)\rangle$$

whereas the only derivation for $loop^\sharp$ is infinite since $q(y)$ will be executed. One consequence of this behaviour is that the control fails to reach the goal $r(v)$ so that the abstract call patterns of $loop^\sharp$ do not safely abstract the concrete call patterns of loop, that is,

$$\text{call}(loop_{\mathsf{go}}) = [\{\langle \mathsf{go}, \varepsilon\rangle, \langle p(x), \varepsilon\rangle, \langle r(x), \varepsilon\rangle\}]_{\sim}$$

$$\text{call}^\sharp(loop^\sharp_{\mathsf{go}}) = [\{\langle \mathsf{go}, true\rangle, \langle p(x), true\rangle, \langle q(x), true\rangle\}]_{\approx}.$$

so that $\text{call}(loop_{\mathsf{go}}) \not\subseteq \gamma(\text{call}^\sharp(loop^\sharp_{\mathsf{go}}))$.

Defining $\alpha[\![c_1, \ldots, c_n]\!] = \alpha_{\text{clause}}[\![c_1]\!], \alpha_{\text{bot}}[\![c_1]\!] \ldots, \alpha_{\text{clause}}[\![c_n]\!], \alpha_{\text{bot}}[\![c_n]\!]$ however, we instead get

go :- $p(v), r(v)$.
go :- $gr(v)$.
$p(w)$:- $(gr(w) \rightarrow (gr(w) \leftrightarrow gr(x))), q(x)$.
$p(w)$:- $gr(w)$.

$q(y)$:- $q(y)$.
$q(y)$:- $gr(y)$.
$r(z)$.
$r(z)$:- $gr(z)$.

The infinite loop caused by executing $q(x)$ is now avoidable and so we get $\langle \mathrm{r(x)}, \varepsilon \rangle \in \mathrm{call}^\sharp(\mathrm{loop}^\sharp_{\mathrm{go}})$. $\square$

Finally, we introduce an operation to relax the propositional formulae obtained in the call patterns. The relax operation is required for technical reasons which are apparent in the proof for the safety of the analysis. For the case where only the property of groundness is analysed, the *relax* operator weakens a propositional formula to a (monotonic) conjunction of ground variables.

Definition 16.

$$relax_V(\phi^\sharp) = \bigwedge \left\{ p_i(v) \;\middle|\; v \in V \wedge \phi^\sharp \models p_i(v) \text{ for } 1 \le i \le n \right\}$$

$$relax([I^\sharp]_\approx) = \left[\left\{ \langle p(\mathbf{x}), relax_{\mathrm{var}(\mathbf{x})}(\phi^\sharp) \rangle \;\middle|\; \langle p(\mathbf{x}), \phi^\sharp \rangle \in I^\sharp \right\} \right]_\approx$$

The following example illustrates the problems that can arise without this weakening of the formulae.

Example 7. Consider the program P listed below on the left together with its corresponding abstract program $P^\sharp$ on the right.

go :- $p(x,y), q(x,y)$.
:- block $p(-,?)$.
$p(x,y)$:- $x = y$.
$q(x,y)$.

go :- $p(x,y), q(x,y)$.
$p(x,y)$:- $(gr(x) \rightarrow gr(y))$.
$q(x,y)$.

With the initial goal go, the atom $p(x,y)$ will be delayed and the call pattern for $q(x,y)$ is $\langle q(x,y), \varepsilon \rangle$. The abstract call pattern is $\langle q(x,y), gr(x) \rightarrow gr(y) \rangle$. Note that even though $assign_{\mathrm{var(q(x,y))}}(\varepsilon) \models gr(x) \rightarrow gr(y)$, due to the downward-closure condition imposed on γ we have $\varepsilon \notin \gamma_{\mathrm{var(q(x,y))}}(gr(x) \rightarrow gr(y))$. Applying the *relax* operation, however, we instead get the abstract call pattern $\langle q(x,y), true \rangle$ and we now have $\varepsilon \in \gamma_{\mathrm{var(q(x,y))}}(true)$. $\square$

We now formally describe an abstract semantics. Like the concrete semantics, this is described in terms of reductions between states. The set of abstract states $State^\sharp$ is defined by $State^\sharp = Goal \times Pos_{Var}$.

Definition 17. For an abstract program $P^\sharp$, the abstract reduction relation $s \rightarrow_{P^\sharp} s'$, where $s, s' \in State^\sharp$ and $s = \langle l :: \mathbf{l}, \phi^\sharp \rangle$, is defined as follows:

- if $l \in Pos_{\mathrm{var(s)}}$ then $s' = \langle \mathbf{l}, \phi^\sharp \wedge l \rangle$;
- if l is $p(\mathbf{x})$ and $h \leftarrow \mathbf{b} \ll_s P^\sharp$ then $s' = \langle \mathbf{b} :: \mathbf{l}, \phi^\sharp \wedge \varphi^\sharp \rangle$ where $\varphi^\sharp = \alpha_{\mathrm{var(h=l)}}(mgu(\{h = l\}))$.

Thus an abstract derivation from an abstract state $s^\sharp$ for an abstract program $P^\sharp$ is an (finite or infinite) sequence of abstract reductions ${s_1}^\sharp \rightarrow_{P^\sharp} {s_2}^\sharp \rightarrow_{P^\sharp} \ldots$ where $s^\sharp = {s_1}^\sharp$. Abstract call patterns can now be defined in terms of the abstract semantics.

Definition 18. $\mathrm{call}^\sharp(\mathrm{P}^\sharp{}_{\mathrm{G}^\sharp}) = [\{\langle \mathrm{l}, \theta^\sharp \restriction \mathrm{var}(\mathrm{l})\rangle \mid \langle \mathrm{G}^\sharp, \mathit{true}\rangle \rightarrow_{P^\sharp}{}^* \langle l :: {G^\sharp}', \theta^\sharp\rangle\}]_\approx$

The theorem below ensures the safety of the analysis. The proof is given in the appendix.

Theorem 19 Safety. If $P^\sharp = \alpha[\![P]\!]$, and $G^\sharp = \alpha_{\mathrm{goal}}[\![G]\!]$ then

$$\mathrm{call}(\mathrm{P_G}) \leq \gamma(\mathit{relax}(\mathrm{call}(\mathrm{P}^\sharp{}_{\mathrm{G}^\sharp})))$$

6 Implementation

An analyser has been implemented in SICStus Prolog which takes, as input, SICStus Prolog programs with block and freeze declarations and produces, as output, call patterns for each predicate in the program. The analyser is largely based on [5] and essentially infers which program variables are bound to ground and non-variable terms. Following [5], Pos formulae are represented as their models and n-ary predicates iff(x, $x_1, \ldots, x_{n-1}$) are used to express formulae of the form $x \leftrightarrow x_1 \wedge \ldots \wedge x_{n-1}$.

Example 8. Consider again the abstract permute program and, in particular, the new and reduce predicates of example 5. In the case of new, each of its clauses are translated into a clause by expressing Pos formulae, like $gr(x') \leftrightarrow (gr(v') \wedge gr(y))$ and $nv(x')$, in terms of iff predicates, like iff(nv(x')) and iff(gr(x'), gr(v'), gr(y)). The single reduce clause, on the other hand, is expanded into three clauses, the formulae $(x_1 \vee x_2) \rightarrow x_3$ being logically equivalent to $(x_1 \wedge x_3) \vee (x_2 \wedge x_3) \vee (\neg x_1 \wedge \neg x_2)$. The neg($x$) predicate expresses $\neg x$. Note that although $\neg x$ is not positive, the model represented by the disjunction of the three clauses is positive. Expressing implications as non-positive formulae merely simplifies the translation of block declarations.

```
new(u, v, w, x) :-
   reduce(new(u,v,w,x), new(u',v',w',x')),
   iff(nv(u'), nv(v')), iff(gr(u'), gr(v')),
   iff(nv(w'), gr(x')), iff(gr(w'), gr(x')).
new(u, v, w, x) :-
   reduce(new(u,v,w,x), new(u',v',w',x')),
   iff(nv(x')),
   iff(gr(x'), gr(v'), gr(y)),
   remove(u', w', y).
new(u, v, w, x) :-
   iff(nv(u)), iff(nv(v)), iff(nv(w)), iff(nv(x)),
   iff(gr(u)), iff(gr(v)), iff(gr(w)), iff(gr(x)).

reduce(new(u,v,w,x), new(u',v',w',x')) :-
   iff(nv(w)),
   iff(nv(u),  nv(u')), iff(gr(u),  gr(u')),
   iff(nv(v),  nv(v')), iff(gr(v),  gr(v')),
   iff(nv(w),  nv(w')), iff(gr(w),  gr(w')),
   iff(nv(x),  nv(x')), iff(gr(x),  gr(x')).
reduce(new(u,v,w,x), new(u',v',w',x')) :-
   iff(nv(x)),
   iff(nv(u),  nv(u')), iff(gr(u),  gr(u')),
   iff(nv(v),  nv(v')), iff(gr(v),  gr(v')),
   iff(nv(w),  nv(w')), iff(gr(w),  gr(w')),
   iff(nv(x),  nv(x')), iff(gr(x),  gr(x')).
reduce(new(u,v,w,x), new(u',v',w',x')) :-
   neg(nv(w)),
   neg(nv(x)).
```

Finally, a magic transform (see, e.g. [5]) coupled with bottom-up evaluation is used to calculate the call patterns of the predicates. □

The benchmarks below are arranged in order according to size. The first program permute.pl is defined earlier in the paper. slowsort.pl (a generate and test algorithm) and interl.pl (simple interpreter for coroutining programs) are NU-Prolog programs with dynamic scheduling written by L. Naish which have been translated into SICStus. nand.pl (a nand-gate circuit designer written by E. Tick) and transp.pl (a matrix transposer written by V. Saraswat) are programs with dynamic scheduling resulting from the automatic translation of concurrent logic programs by the QD-Janus system [10]. Dynamic scheduling is used to emulate the concurrency present in the original programs. The programs send.pl (a cryptoarithmetic problem) and queens.pl (a coroutining n-queens) were provided by M. Carlsson. Finally primes.pl (a lazy primes sieve) and permcon.pl (a monotonic permutation generator) have been written by ourselves.

Program	*Cl*	*Lit*	*DL*	*Analysis Time (Secs)*
permute.pl	5	4	1	0.00
send.pl	6	13	4	0.01
slowsort.pl	9	8	8	0.01
primes.pl	10	12	8	0.01
interpl.pl	11	10	3	0.01
queens.pl	12	17	1	0.01
permcon.pl	21	28	7	0.03
nand.pl.pl	90	157	13	0.41
transp.pl	112	180	20	1.10

Cl is the number of clauses analysed, Lit is the number of literals and DL is the number of delaying literals. The analysis times are based on a Pentium 200MHz PC machine with 64M of memory running SICStus prolog version 3.5 under Linux where the analyser is compiled into compact code (byte code).

Our times indicate an efficient analysis for small and medium-sized programs. This is in contrast for the times reported in [19] for a groundness and non-variable analysis based on the more general closure framework of [14]. Although the results given in [19] were mostly acceptable, there were some anomalous times. In particular the time quoted for transp.pl was 168.5 seconds compared to 1.10 seconds using our analyser. (Note that this is not an exact comparison since the times are based on different machines.)

7 Related Work

Suspension-analysis framework The simple and practical framework of [9], though adequate for inferring suspension-freeness – its primary objective – it does not provide a suitable basis for accurately tracing properties like groundness since it abstracts the behaviour of possibly delaying goals in a very conservative way. Even when equipped with a Pos domain (rather than a traditional mode domain [9]) the analysis cannot infer modes to the precision of our framework.

Multiset framework The multiset framework of [17] uses a multiset to record which atoms are definitely delayed and which atoms are possibly delayed. However, in [14] (which builds on and improves the method) it is reported that "the analysis is imprecise and rather inefficient in practice".

Closure framework In an attempt to alleviate some of the problems with the multiset framework, a closure based semantics is proposed in [14]. The semantics improves the precision but efficient implementation is still difficult. For example, for the approach to be practical, the implementation described in [14] makes "observational equivalence" of closures, represents closures in a special way, and uses a number of techniques, including the differential approach [13], to improve the handling of cases when no atoms are delayed. Furthermore, this framework is more complicated than necessary for applications like determinacy analysis since the framework is aimed at tracing properties such as definite freeness that are not downward closed. Our work adapts and simplifies this work for applications like determinacy analysis. In [19], the closure framework is used to underpin two program transformations: one that simplifies delay conditions, and another that reorders delayed literals.

Abstracting synchronisation framework Our transform, which *encodes* synchronisation, is not to be confused with the $NoSynch$ transform of [21] which *removes* synchronisation from a concurrent constraint program. The work of [21] shows how analyses developed for constraint programs can be used to reason about non-suspending concurrent constraint programs.

Abstract compilation frameworks Our analysis blends the closure framework of [14] with the abstract compilation work of [5] and [11]. Abstract compilation and abstract programs have been proposed for deriving modes [5, 7], encoding regular approximations [12], inferring directional types [4] and deducing inter-argument relationships [15]. In addition, [5, 16], [15] and [7] present abstract compilation schemes in which analysis is realised as the bottom-up and top-down evaluation of constraint logic programs. None of these approaches, however, traces synchronisation behaviour.

8 Future work

Future work will focus on generalising the technique to other downwards-closed domains and then later to non downwards-closed domains. Another direction would be to investigate how demand-driven analysis can be used to infer the modes that suspending goals are reawoken with. We also plan to adapt widening techniques from [2] and investigate their effect on our method.

Acknowledgments

This work was supported by EPSRC Grant GR/K79635 and partly by the Nuffield Grant SCI/180/94/417/G. We gratefully acknowledge Brandon Bennett, Mats Carlsson, Jan Smaus and Andy Verden for their helpful discussions. Also we thank Roberto Bagnara, Mike Codish and Germán Puebla for help with the implementation and benchmarking.

References

1. R. Bagnara. *Data-Flow Analysis for Constraint Logic-Based Languages*. PhD thesis, University of Pisa, 1997.
2. F. Benoy, M. Codish, A. Heaton, and A. King. Widening Pos for Efficient and Scalable Groundness Analysis of Logic Programs. Submitted for Publication.
3. M. Carlsson. Personal communication on the `freeze/2` and `block` declaration awakening order in SICStus Prolog 3. April 1996.
4. M. Codish and B. Demoen. Deriving polymorphic type dependencies for logic programs using multiple incarnations of prop. In *SAS'94*, pages 281–297. Springer-Verlag, 1994.
5. M. Codish and B. Demoen. Analysing Logic Programs using "prop"-ositional Logic Programs and a Magic Wand. *Journal of Logic Programming*, 25(3):249–274, 1995.
6. M. Codish, M. Falaschi, and K. Marriott. Suspension analyses for concurrent logic programs. *ACM Transactions on Programming Languages and Systems*, 16(3):649–686, 1994.
7. P. Codognet and G. Filé. Computations, Abstractions and Constraints. Technical report, Dipartimento di Matematica Pura e Applicata, Università di Padova, 1991.
8. A. Cortesi, G. Filé, and W. Winsborough. Optimal Groundness Analysis using Propositional Logic. *Journal of Logic Programming*, 27(1, 2 and 3):137–169, 1996.
9. S. Debray, D. Gudeman, and P. Bigot. Detection and Optimization of Suspension-free Logic Programs. In *ILPS'94*, pages 487–504. MIT Press, 1994.
10. S.K. Debray. QD-Janus: A Sequential Implementation of Janus in Prolog. *Software- Practice and Experience*, 23(12):1337–1360, 1993.
11. J. Gallagher, D. Boulanger, and H. Saglam. Practical Model-Based Static Analysis for Definite Logic Programs. Technical Report CSTR-95-011, University of Bristol, 1995.
12. J. Gallagher and A. de Waal. Fast and precise regular approximations of logic programs. In *ICLP'94*, pages 599–613. MIT Press, 1994.
13. M. García de la Banda, K. Marriott, H. Søndergaard, and P. Stuckey. Improved Analysis of Logic Programs using a Differential Approach. Technical Report 95/20, Computer Science Department, Melbourne University, May 1995.
14. M. García de la Banda, K. Marriott, and P. Stuckey. Efficient Analysis of Logic Programs with Dynamic Scheduling. In *ILPS'95*, pages 417–431. MIT Press, 1995.
15. R. Giacobazzi, S. K. Debray, and G. Levi. A generalised semantics for constraint logic programs. In *FGCS'92*, pages 581–591. ACM Press, 1992.
16. M. Hermenegildo, R. Warren, and S. K. Debray. Global flow analysis as a practical compilation tool. *J. Logic Programming*, 13(1, 2, 3 and 4):349–366, 1992.
17. K. Marriott, M. García de la Banda, and M. Hermenegildo. Analyzing Logic Programs with Dynamic Scheduling. In *POPL'94*, pages 240–253. ACM Press, 1994.
18. K. Marriott and H. Søndergaard. Precise and efficient groundness analysis. *ACM Lett. Program. Lang. Syst.*, 13(2–3):181–196, 1993.
19. G. Puebla, M. García de la Banda, K. Marriott, and P. Stuckey. Optimization of Logic Programs with Dynamic Scheduling. In *ICLP'97*. MIT Press, 1997.
20. A. Verden. Personal communication on `freeze/2` and the `const_delay` module of IF/Prolog V5.0. April 1996.
21. E. Zaffanella, R. Giacobazzi, and G. Levi. Abstracting Synchronisation in Concurrent Constraint Programming. In *PLILP'94*, pages 57–72. Springer-Verlag, 1994.

A Proof for Safety

The first stage of the proof is to define semi-abstract reduction sequences. These are sequences where the selection rule of the concrete derivation is preserved. Hence each abstract state for a semi-abstract reduction sequence requires a set to store delayed atoms (as in the concrete case). The next stage is to define a reordering procedure for the semi-abstract reduction sequences. This allows us to obtain the "full" abstract reduction sequences.

Note that for $\alpha_{\text{clause}}[\![c_1]\!]$, $\alpha_{\text{goal}}[\![\mathbf{b}]\!]$, ... we omit the subscript to instead write $\alpha(c_1)$, $\alpha(\mathbf{b})$, ...

Definition 20 labelled reductions. Reductions are labelled $s {\rightarrow_P}^q s'$ as follows:

- if $s = \langle l :: G, \theta, D\rangle$, l is $x = t$ and $s' = \langle D' :: G, \theta', D \setminus D'\rangle$ then $s {\rightarrow_P}^{(r(l),D')} s'$;
- if $s = \langle l :: G, \theta, D\rangle$ and l is an atom such that $reduce(l, \theta)$ holds and $s' = \langle \mathbf{b} :: G, \theta', D\rangle$ then $s {\rightarrow_P}^{(r(l),\mathbf{b})} s'$;
- if $s = \langle l :: G, \theta, D\rangle$ and l is an atom such that $reduce(l, \theta)$ fails then $s {\rightarrow_P}^{(d(l),\epsilon)} s'$;

Definition 21 derived selection rule for a sequence of reductions. Given a sequence of reductions $s_1 {\rightarrow_P} \ldots {\rightarrow_P} s_n$, the derived selection rule is $q_1, \ldots, q_{n-1}$ where for each $1 \leq i \leq n-1$, $s_i {\rightarrow_P}^{q_i} s_{i+1}$.

Definition 22 semi-abstract reduction sequences. Suppose $s_1 {\rightarrow_P} \ldots {\rightarrow_P} s_n$ is a sequence of reductions with the derived selection rule $q_1, \ldots q_{n-1}$. Then the semi-abstract sequence of reductions $s'_1 {\Rightarrow_{P^\sharp}} \ldots {\Rightarrow_{P^\sharp}} s'_n$ is defined such that $s'_1 = \langle \alpha(G_1), true, \epsilon\rangle$, and for each $1 \leq i \leq n-1$, letting $s_i = \langle \alpha(G_i), \phi^\sharp_i, \alpha(D_i)\rangle$ then $s_{i+1} = \langle \alpha(G_{i+1}), \phi^\sharp_{i+1}, \alpha(D_{i+1})\rangle$ where:

- if $q_i = (r(l), D)$, $\alpha(G_i) = \alpha(l) :: \alpha(G'_i)$ and $\alpha(l) \in Pos_{\mathrm{var(s)}}$ then $\alpha(G_{i+1}) = \alpha(D) :: \alpha(G'_i)$, $\phi^\sharp_{i+1} = \phi^\sharp_i \wedge \alpha(l)$, and $\alpha(D_{i+1}) = \alpha(D_i) \setminus \alpha(D)$;
- if $q_i = (r(l), \mathbf{b})$, $\alpha(G_i) = \alpha(l) :: \alpha(G'_i)$, $\alpha(l)$ is $p(\mathbf{x})$ and $p(\mathbf{y}) \leftarrow (\phi^\sharp_r \rightarrow \phi^\sharp_s), \alpha(\mathbf{b}) \ll_s P^\sharp$ then $\alpha(G_{i+1}) = \alpha(\mathbf{b}) :: \alpha(G'_i)$, $\phi^\sharp_{i+1} = \phi^\sharp \wedge \varphi^{\sharp'} \wedge (\phi^\sharp_r \rightarrow \phi^\sharp_s)$ and $\alpha(D_{i+1}) = \alpha(D_i)$ where $\varphi^{\sharp'} = \alpha_{\mathrm{var(p(\mathbf{x})=p(\mathbf{x}'))}}(mgu(\{p(\mathbf{x}) = p(\mathbf{x}')\}))$, $\phi^\sharp_r = \bigvee_{reduce^\sharp(p(\mathbf{x}'),\phi^\sharp)} \phi^\sharp$, $\phi^\sharp_s = \alpha^\sharp_{\mathrm{var}(\mathbf{x}'=\mathbf{y})}(\mathrm{mgu}(\{\mathbf{x}' = \mathbf{y}\}))$, and also $\mathrm{var(s_i)}$, $\mathrm{var}(\mathbf{x}')$ and $\mathrm{var(p(\mathbf{y}) \leftarrow \mathbf{b})}$ are all distinct from each other.
- if $q_i = (d(l), \epsilon)$, $\alpha(G_i) = \alpha(l) :: \alpha(G'_i)$ and $\alpha(l)$ is $p(\mathbf{x})$ then $\alpha(G_{i+1}) = \alpha(G'_i)$, $\phi^\sharp_{i+1} = \phi^\sharp_i$ and $\alpha(D_{i+1}) = \alpha(D_i) \cup \{\alpha(l)\}$.

Lemma 23. Let $s_1 {\rightarrow_P} \ldots {\rightarrow_P} s_n$ be a sequence of concrete reductions such that $s_i = \langle G_i, \theta_i, D_i\rangle$ for each $1 \leq i \leq n$. Then in the sequence of semi-abstract reductions $s'_1 {\Rightarrow_P} \ldots {\Rightarrow_P} s'_n$ we have $s'_i = \langle \alpha(G_i), \phi^\sharp_i, \alpha(D_i)\rangle$ and also for every $1 \leq i \leq n$ $assign_{\mathrm{var(s_i)}}(\theta_i) \models \phi^\sharp_i \restriction \mathrm{var(s_i)}$.

Proof. The proof is by induction on i.

For the base case we have $i = 1$ and $s_1 = \langle G_1, \varepsilon, \epsilon\rangle$, $s'_1 = \langle \alpha(G_1), true, \epsilon\rangle$. But $assign_{\mathrm{var(s_1)}}(\varepsilon) \models true$ and so the result holds.

Next consider the induction step. First suppose that $G_i = l_i :: G'_i$ where l_i is an equation $x = t$. Here we have $s_i = \langle x = t :: G'_i, \theta_i, D_i\rangle \rightarrow_P \langle D :: G'_i, \theta', D_i \setminus D\rangle$ where $\theta' \in \mathrm{mgu}(\{\mathrm{x = t}\} \cup \mathrm{eqn}(\theta_\mathrm{i}))$. Using the induction hypothesis there is a state $s'_i = \langle \alpha(x = t :: G_i), \phi^\sharp_i, \alpha(D_i)\rangle$ with $assign_{\mathrm{var(s_i)}}(\theta_i) \models \phi^\sharp_i \restriction \mathrm{var(s_i)}$. Now $s'_i \Rightarrow_{P^\sharp} \langle \alpha(D :: G_i), \phi^\sharp_i \wedge \alpha(l_i), \alpha(D_i) \setminus \alpha(D)\rangle$, and also we have $\alpha(l_i) = \alpha_{\mathrm{var(x=t)}}(\mathrm{mgu}(\{\mathrm{x = t}\}))$. But

$\mathrm{mgu}(\{\mathrm{x = t}\}) \subseteq \gamma_{\mathrm{var(x=t)}}(\alpha_{\mathrm{var(x=t)}}(\mathrm{mgu}(\{\mathrm{x = t}\})))$

and so for $\varphi \in \mathrm{mgu}(\{\mathrm{x = t}\})$ we get $assign_{\mathrm{var(x=t)}}(\varphi) \models \alpha_{\mathrm{var(x=t)}}(\mathrm{mgu}(\{\mathrm{x = t}\}))$. Now $\mathrm{var(s_i)} = \mathrm{var(s_{i+1})}$, and so it follows that for $\theta' \in \mathrm{mgu}(\{\mathrm{x = t}\} \cup \mathrm{eqn}(\theta_\mathrm{i}))$ we get $assign_{\mathrm{var(s_{i+1})}}(\theta') \models \phi^\sharp_i \wedge \alpha_{\mathrm{var(x=t)}}(\mathrm{mgu}(\{\mathrm{x = t}\})) \restriction \mathrm{var(s_i)}$, as required. (This last step corresponds to the safety of abstract unification for the Pos domain. See, e.g. [8] for details).

Next suppose that l_i is an atom $p(\mathbf{x})$ and l_i suspends, i.e. $s_i = \langle l_i :: G'_i, \theta_i, D_i\rangle \rightarrow_{P^\sharp} s_{i+1} = \langle G'_i, \theta_i, D_i \cup \{l_i\}\rangle$. By the induction hypothesis, $s'_i = \langle \alpha(l_i :: G_i), \phi^\sharp_i, \alpha(D_i)\rangle$ with $assign_{\mathrm{var(s_i)}}(\theta_i) \models \phi^\sharp_i \restriction \mathrm{var(s_i)}$. But $s'_i \Rightarrow_{P^\sharp} \langle \alpha(G_i), \phi^\sharp_i, \alpha(D_i) \cup \{\alpha(l_i)\}\rangle$, and since $\mathrm{var(s_i)} = \mathrm{var(s_{i+1})}$ the result follows.

Finally, suppose that l_i is an atom $p(\mathbf{x})$ and reduces, i.e. $s_i = \langle l_i :: G'_i, \theta_i, D_i\rangle \rightarrow_{P^\sharp} \langle \mathbf{b} :: G'_i, \theta', D_i\rangle$ where $\theta' \in \mathrm{mgu}(\{\mathrm{h = l}\} \cup \mathrm{eqn}(\theta_\mathrm{i}))$, $h = p(\mathbf{y})$ and $p(\mathbf{y}) \leftarrow \mathbf{b} \ll_s P^\sharp$. Now $s'_i \Rightarrow_{P^\sharp} \langle \alpha(\mathbf{b} :: G'_i), \phi^\sharp_i \wedge \varphi^\sharp \wedge (\phi^\sharp_r \rightarrow \phi^\sharp_s), \alpha(D_i)\rangle$, where $\varphi^\sharp = \alpha_{\mathrm{var(p(\mathbf{x})=p(\mathbf{x}'))}}(mgu(\{p(\mathbf{x}) = p(\mathbf{x'})\}))$ and $\phi^\sharp_r = \bigvee_{reduce^\sharp(p(\mathbf{x}'), \phi^\sharp)} \phi^\sharp$ and $\phi^\sharp_s = \alpha^\sharp_{\mathrm{var}(\mathbf{x}'=\mathbf{y})}(\mathrm{mgu}(\{\mathbf{x'} = \mathbf{y}\}))$, and also $\mathrm{var}(\mathbf{x}')$, $\mathrm{var(s'_j)}$ and $\mathrm{var(p(\mathbf{y}) \leftarrow \mathbf{b})}$ are all distinct variables. We have $assign_{\mathrm{var(s_i)}}(\theta_i) \models \phi^\sharp_i$ by the induction hypothesis. Can show that $\mathrm{mgu}(\{\mathrm{h = l}\}) \subseteq \gamma_{\mathrm{var(s_i)}}((\varphi^\sharp \wedge \phi^\sharp_\mathrm{s}) \restriction \mathrm{var(s_i)})$ and hence $assign_{\mathrm{var(s_i)}}(\psi) \models (\varphi^\sharp \wedge \phi^\sharp_s) \restriction \mathrm{var(s_i)}$ for $\psi \in \mathrm{mgu}(\{\mathrm{h = l}\})$. Hence from the induction hypothesis, $assign_{\mathrm{var(s_i)}}(\theta') \models (\phi^\sharp_i \wedge \varphi^\sharp \wedge \phi^\sharp_s) \restriction \mathrm{var(s_i)}$ since $\theta' \in \mathrm{mgu}(\{\mathrm{h = l}\}) \cup \mathrm{eqn}(\theta_\mathrm{i}))$. Noting that $a \rightarrow b \equiv \neg a \vee b$, it follows that $assign_{\mathrm{var(s_i)}}(\theta') \models (\phi^\sharp_i \wedge \varphi^\sharp \wedge (\phi^\sharp_r \rightarrow \phi^\sharp_s)) \restriction \mathrm{var(s_i)}$ as required.

Definition 24 reawoken goals. Suppose $s_1 \rightarrow_{P^\sharp} \ldots \rightarrow_{P^\sharp} s_n$ is a sequence of semi-abstract reductions with selection rule $q_1, \ldots, q_{n-1}$. An atom a is reawoken if there exists $1 \leq i < j \leq n$ such that $q_i = (d(a), \epsilon)$ and $q_j = (r(a), A)$ for some A.

Lemma 25. Let $s = \langle G :: G', \phi^\sharp, D\rangle$ be an abstract state. Then there is a sequence of abstract reductions such that $s \rightarrow_{P^\sharp} \ldots \rightarrow_{P^\sharp} s' = \langle G', \phi^{\sharp'}, D\rangle$ for some $\phi^{\sharp'}$.

Proof. Follows easily since every abstract atom $p(\mathbf{x})$ has a rule $p(\mathbf{x}) \leftarrow \phi^\sharp_b$ defining it.

Definition 26 procedure for reordering reductions. Let $s_1 \Rightarrow_{P^\sharp} \ldots \Rightarrow_{P^\sharp} s_n$ be a sequence of semi-abstract reductions labelled by $q_1, \ldots, q_{n-1}$, and suppose there are m reawoken goals. The procedure $R(m)$ is defined inductively according to the number of reawoken goals m as follows:

– If there are no reawoken goals then the sequence remains the same.

– If there are $m + 1$ reawoken goals then suppose the procedure $R(m)$ for m reawoken goals is already defined. Let $s_j = \langle l :: G, \phi^\sharp, D\rangle$ be the state such that $q_j = (r(l), A)$ where l is the last reawoken goal in $s_1 \Rightarrow_{P^\sharp} \ldots \Rightarrow_{P^\sharp} s_n$, and suppose $\exists j_0 \leq n$ such that $s_j \Rightarrow_{P^\sharp} \ldots \Rightarrow_{P^\sharp} s_{j_0} = \langle G, \phi^{\sharp'}, D'\rangle$. Let $q_1^m, \ldots, q_{n_0}^m$ be the selection rule obtained by applying $R(m)$ to $s'_1 \Rightarrow_{P^\sharp} \ldots \Rightarrow_{P^\sharp} s'_j$, where the selection rule giving this sequence is the same as $q_1, \ldots, q_{n-1}$ except that it does not reawake l. Letting $q_k^m = (d(l), \epsilon)$, replace the selection rule $q_1^m, \ldots, q_{n_0}^m$ with the new selection rule

  $q_1^m, \ldots, q_{k-1}^m, q_j, \ldots, q_{j_0-1}, q_{k+1}^m, \ldots, q_{n_0}^m, q_{j_0}, \ldots, q_{n-1}.$

– For the case above suppose there is no $j_0 \leq n$ such that $s_j \Rightarrow_{P^\sharp} \ldots \Rightarrow_{P^\sharp} s_{j_0} = \langle G, \phi^{\sharp'}, D'\rangle$. As above, let $q_1^m, \ldots, q_{n_0}^m$ be the selection rule obtained by applying $P(m)$ to $s'_1 \Rightarrow_{P^\sharp} \ldots \Rightarrow_{P^\sharp} s'_j$. Let $s_n \rightarrow_{P^\sharp} \ldots \rightarrow_{P^\sharp} s_{j_1+1}$ be a sequence of reductions (obtained using the α_{bot} clauses) such that $s_{j_1+1} = \langle G, \phi^{\sharp'}, D'\rangle$, and also let $q_n, \ldots, q_{j_1}$ be the derived selection rule for this sequence. Letting $q_k^m = (d(l), \epsilon)$, replace the selection rule $q_1^m, \ldots, q_{n_0}^m$ with the new selection rule

  $q_1^m, \ldots, q_{k-1}^m, q_j, \ldots, q_{j_1}, q_{k+1}^m, \ldots, q_{n_0}^m.$

Proposition 27. The process above terminates with the new selection rule containing no reawoken goals.

Definition 28. Let $s_1 \rightarrow_P \ldots \rightarrow_P s_n$ be a sequence of concrete reductions with $s_n = \langle l_n :: G_n, \theta_n, D_n\rangle$. Also suppose there is a state $s_i = \langle l_i :: G_i, \theta_i, D_i\rangle$ such that $s_i \rightarrow_P \langle \mathbf{b} :: G', \theta', D'\rangle$ and $l_n \in \mathbf{b}$. Then for any literal l, l is a parent atom of l_n if either $l = l_i$ or l is a parent atom of l_i.

Definition 29. Let $s_1 \rightarrow_P \ldots \rightarrow_P s_n$ be a sequence of concrete reductions with $s_n = \langle l_n :: G_n, \theta_n, D_n\rangle$. Then s_n is in a reawoken part of $s_1 \rightarrow_P \ldots \rightarrow_P s_n$ iff l_n has a parent atom which is reawoken. Otherwise we say that s_n is in a non-reawoken part of $s_1 \rightarrow_P \ldots \rightarrow_P s_n$.

Lemma 30. Suppose $\phi^\sharp$ is a monotonic formula. Then $assign_V(\theta) \models \phi^\sharp \Rightarrow \theta \in \gamma_V(\phi^\sharp)$.

Proof. In [8] it is shown that $assign_V(\theta) \models \phi^\sharp \Rightarrow assign_V(\theta') \models \phi^\sharp$ for all $\theta' \leq \theta$ when $\phi^\sharp$ is monotonic. Hence result follows from the definition of $\gamma_V(\phi^\sharp)$.

Lemma 31. Let $s_1 \rightarrow_P \ldots \rightarrow_P s_n$ be a sequence of concrete reductions with m reawoken goals, and suppose that $s_i = \langle l_i :: G_i, \theta_i, D_i\rangle$ is a state in a non-reawoken part. Then there is a state $s'_{i_0} = \langle \alpha(l_i) :: G'_i, \phi^\sharp_{i_0}, D'_{i_0}\rangle$ in the sequence of semi-abstract reductions $s'_1 \Rightarrow_{P^\sharp} \ldots \Rightarrow_{P^\sharp} s'_{n_0}$ given by $R(m)$, $assign_{\mathrm{var(l_i)}}(\theta_i) \models relax_{\mathrm{var(l_i)}}(\phi^\sharp_{i_0})$.

Proof. By lemma 23 there is a semi-abstract sequence of reductions with the same selection rule as the concrete sequence of reductions and also with $assign_{\mathrm{var(s_i)}}(\theta_i) \models \phi^\sharp_i \restriction \mathrm{var(s_i)}$ for each i. By induction on the number of reawoken goals m, we show that

$assign_{\mathrm{var(s_i)}}(\theta_i) \models \phi^\sharp_{i_0} \restriction \mathrm{var(s_i)}$ in the semi-abstract sequence given by $R(m)$. Since $\phi^\sharp_{i_0} \restriction \mathrm{var(s_i)} \models relax_{\mathrm{var(l_i)}}(\phi^\sharp_{\mathrm{i_0}})$, the result then follows.

If the are no reawoken goals then the semi-abstract sequence remains unchanged and so the result holds.

Suppose there are $m+1$ reawoken goals. Let $s_j = \langle l :: G, \phi^\sharp, D\rangle$ be the state such that $q_j = (r(l), A)$ where l is the last reawoken goal in $s_1 \Rightarrow_{P^\sharp} \ldots \Rightarrow_{P^\sharp} s_n$, and suppose $\exists j_0 \leq n$ such that $s_j \Rightarrow_{P^\sharp} \ldots \Rightarrow_{P^\sharp} s_{j_0} = \langle G, \phi^{\sharp'}, D'\rangle$. Let $q^m_1, \ldots, q^m_{n_0}$ be the selection rule obtained by applying $R(m)$ to $s'_1 \Rightarrow_{P^\sharp} \ldots \Rightarrow_{P^\sharp} s'_j$. Letting $q^m_k = (d(l), \epsilon)$, the new selection rule is $q^m_1, \ldots, q^m_{k-1}, q_j, \ldots, q_{j_0-1}, q^m_{k+1}, \ldots, q^m_{n_0}, q_{j_0}, \ldots, q_{n-1}$. Let s_i be a concrete state in a non-reawoken part. First suppose $i \leq j$. By the induction hypothesis there is a corresponding state $s'_{i_0} = \langle \alpha(l_i) :: G', \phi^\sharp_{i_0}, D'\rangle$ in the semi-abstract sequence of reductions $s^m_1 \Rightarrow_{P^\sharp} \ldots \Rightarrow_{P^\sharp} s^m_{n_0}$ given by $R(m)$ with $assign_{\mathrm{var(s_i)}}(\theta_i) \models \phi^\sharp_{i_0}$. Consider the selection rule given by $R(m+1)$. For every $1 \leq i' \leq k$, we have $s^{m+1}_{i'} = s^m_{i'}$. Hence the result holds for this case by the induction hypothesis. Let $l = p(\mathbf{x})$. We have

$$s^{m+1}_k = \langle \alpha(l) :: G', \phi^\sharp_k, D'\rangle \Rightarrow_{P^\sharp} \ldots \Rightarrow_{P^\sharp}$$

$$s^{m+1}_{k+j_0-j} = \langle G', \phi^\sharp_k \wedge \varphi^{\sharp'} \wedge (\phi^\sharp_r \rightarrow \phi^\sharp_s) \wedge \phi^\sharp_b, D''\rangle$$

for some $\phi^\sharp_b$, where $\varphi^\sharp = \alpha_{\mathrm{var(p(\mathbf{x})=p(\mathbf{x'}))}}(mgu(\{p(\mathbf{x}) = p(\mathbf{x'})\}))$ and $\phi^\sharp_r = \bigvee_{reduce^\sharp(p(\mathbf{x'}),\phi^\sharp)} \phi^\sharp$ and $\phi^\sharp_s = \alpha^\sharp_{\mathrm{var(\mathbf{x'}=\mathbf{y})}}(\mathrm{mgu}(\{\mathbf{x'} = \mathbf{y}\}))$, and also $\mathrm{var}(\mathbf{x'})$, $\mathrm{var(s'_j)}$ and $\mathrm{var(p(\mathbf{y}) \leftarrow \mathbf{b})}$ are all distinct variables. It follows that for all $k < i' \leq i$ we have

$$s^{m+1}_{i'+j_0-j} = \langle l_{i'} :: G_{i'}, \phi^\sharp_{i'} \wedge \varphi^{\sharp'} \wedge (\phi^\sharp_r \rightarrow \phi^\sharp_s) \wedge \phi^\sharp_b, D_{i'}\rangle$$

where $s^m_{i'} = \langle l_{i'} :: G_{i'}, \phi^\sharp_{i'}, D'_{i'}\rangle$ for some $D'_{i'}$. Now for every concrete state s_i in a non-reawoken part, $reduce(l_i, \theta_i)$ fails. Hence it is sufficient to show that

$$assign_{\mathrm{var(s_i)}}(\theta_i) \models \phi^\sharp \Rightarrow assign_{\mathrm{var(s_i)}}(\theta_i) \models \phi^\sharp \wedge \varphi^{\sharp'} \wedge (\phi^\sharp_r \rightarrow \phi^\sharp_s) \wedge \phi^\sharp_b$$

whenever $reduce(l_i, \theta_i)$ fails.

Let $\varphi^\sharp_r = \bigvee_{reduce^\sharp(p(\mathbf{x}),\phi^\sharp)} \phi^\sharp$, and suppose that $assign_{\mathrm{var(s_i)}}(\theta_i) \models \varphi^\sharp_r$. Then $assign_{\mathrm{var(l_i)}}(\theta_i) \models \varphi^\sharp_r$ also. Since $\varphi^\sharp_r$ is monotonic, we get $\theta_i \in \gamma_{\mathrm{var(l_i)}}(\phi^\sharp)$. But $reduce^\sharp(l_i, \varphi^\sharp_r)$ holds and so we must have $reduce(l_i, \theta_i)$ holds. This is a contradiction, and so we cannot have $assign_{\mathrm{var(s_i)}}(\theta_i) \models \varphi^\sharp_r$. Since $\mathrm{var}(\varphi^\sharp_{\mathrm{r}}) \subseteq \mathrm{var(s_i)}$ it follows that $assign_{\mathrm{var(s_i)}}(\theta_i) \models \neg\varphi^\sharp_r$. Hence we also get $assign_{\mathrm{var(s_i)}}(\theta_i) \models (\neg\phi^\sharp_r \wedge \varphi^\sharp) \restriction \mathrm{var(s_i)}$. Therefore we have $assign_{\mathrm{var(s_i)}}(\theta_i) \models (\phi^\sharp \wedge \neg\phi^\sharp_r \wedge \varphi^\sharp) \restriction \mathrm{var(s_i)}$. Now since $\mathrm{var}(\phi^\sharp_{\mathrm{b}})$ are distinct from both $\mathrm{var(s_i)}$ and $\mathrm{var}(\varphi^\sharp)$, it also follows that $assign_{\mathrm{var(s_i)}}(\theta_i) \models (\phi^\sharp \wedge \neg\phi^\sharp_r \wedge \varphi^\sharp \wedge \phi^\sharp_b) \restriction \mathrm{var(s_i)}$. Hence noting that $a \rightarrow b \equiv \neg a \vee b$, we obtain $assign_{\mathrm{var(s_i)}}(\theta_i) \models (\phi^\sharp \wedge (\phi^\sharp_r \rightarrow \phi^\sharp_s) \wedge \varphi^\sharp \wedge \phi^\sharp_b) \restriction \mathrm{var(s_i)}$ as required.

Next suppose that $i > j$. Since s_i is in a non-reawoken part we must have $i > j_0$. Consider the corresponding state $s^{m+1}_{i_1}$ for s_i in the sequence of reductions given by $R(m+1)$, i.e. the state with $q^{m+1}_{i_1} = q(i)$. The goals reduced in the sequence $s^{m+1}_1 \Rightarrow_{P^\sharp} \ldots \Rightarrow_{P^\sharp} s^{m+1}_{i_1}$ will be the same as those reduced in the semi-abstract sequence of reductions with the selection rule $q_1, \ldots, q_{n-1}$. Hence this corresponding state will be the same as the corresponding state in the semi-abstract sequence of reductions with the same selection rule as the concrete sequence of reductions. Hence the result follows from lemma 23.

For the third case (of definition 26), the proof follows similarly to the above.

Lemma 32. Let $s_1 \rightarrow_P \ldots \rightarrow_P s_n$ be a sequence of concrete reductions such that $s_n = \langle l :: G, \theta, D\rangle$ with s_n in a non-reawoken part. Then there is a sequence of abstract reductions $s'_1 \rightarrow_{P^\sharp} \ldots \rightarrow_{P^\sharp} s'_m$ such that $s'_m = \langle \alpha(l) :: G', \phi^\sharp\rangle$ and $assign_{\mathrm{var(l)}}(\theta) \models relax_{\mathrm{var(l)}}(\phi^\sharp)$.

Proof. Suppose $l' \in D$. Hence for all $1 \leq i \leq n$, $reduce(l', \theta_i)$ fails where $s_i = \langle G_i, \theta_i, D_i\rangle$. Let s_j be the state such that $q_j = (d(l'), \epsilon)$. In the semi-abstract sequence of reductions obtained using lemma 31 replace q_j with q'_j, q''_j where $q'_j = (r(l'), \phi^\sharp_b)$ (where $l' \leftarrow \phi^\sharp_b \ll_s P^\sharp$), and $q''_j = (r(\phi^\sharp_b), \epsilon)$. We can now show that $assign_{\mathrm{var(l)}}(\theta) \models relax_{\mathrm{var(l)}}(\phi^\sharp)$ with the new semi-abstract sequence of reductions using a similar argument to that found in the proof of lemma 31. Repeating this process, we end up with a semi-abstract sequence of reductions without any atoms being delayed. This is equivalent to having a sequence of "full" abstract reductions.

Lemma 33. Let $s_1 \rightarrow_P \ldots \rightarrow_P s_n$ be a sequence of concrete reductions such that $s_n = \langle l :: G, \theta, D\rangle$ where s_n is in a reawoken part. Then there is a sequence of semi-abstract reductions $s'_1 \rightarrow_{P^\sharp} \ldots \rightarrow_{P^\sharp} s'_m$ such that $s'_m = \langle \alpha(l) :: G', \phi^\sharp\rangle$ and $assign_{\mathrm{var(l)}}(\theta) \models relax_{\mathrm{var(l)}}(\phi^\sharp)$.

The proof (omitted here) essentially involves defining a procedure giving a reordering of the concrete sequence of reductions similar to that given by the procedure $R(m)$ for the semi-abstract reduction sequences. This allows us to obtain a new concrete substitution θ' such that θ is an instance of θ' and $assign_{\mathrm{var(l)}}(\theta') \models \phi^\sharp$. Given this, it then follows that $assign_{\mathrm{var(l)}}(\theta) \models relax_{\mathrm{var(l)}}(\phi^\sharp)$ as required.

Theorem 34 Safety. If $P^\sharp = \alpha[\![P]\!]$, and $G^\sharp = \alpha_{\mathrm{goal}}[\![G]\!]$ then

$$\mathrm{call}(\mathrm{P_G}) \leq \gamma(relax(\mathrm{call}(\mathrm{P^\sharp}_{\mathrm{G^\sharp}})))$$

Proof. If $\langle l, \theta \restriction \mathrm{var(l)}\rangle \in \gamma(relax(\mathrm{call}(\mathrm{P^\sharp}_{\mathrm{G^\sharp}})))$ then exists $\langle \alpha(l), \phi^\sharp \restriction \mathrm{var(l)}\rangle \in \mathrm{call}(\mathrm{P^\sharp}_{\mathrm{G^\sharp}})$ such that $\theta \in \gamma_{\mathrm{var(l)}}(relax_{\mathrm{var(l)}}(\phi^\sharp))$. Since $relax_{\mathrm{var(l)}}(\phi^\sharp)$ is monotonic this is equivalent to having $assign_{\mathrm{var(l)}}(\theta) \models relax_{\mathrm{var(l)}}(\phi^\sharp)$. Hence the result follows from lemmas 31 and 33.