Practical Analysis of Logic Programs with Delay (Extended Abstract)

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. A simple prototype implementation has been built applying some of these methods to our analysis, and at this initial stage has achieved accurate results in promising times.

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.

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 \(\text{block new}\_\text{remove}(?, ?, -,-, -)\), the “-” in the third and fourth argument positions means that in a call to\new\_\text{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\_\text{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\_\text{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 back-tracking.

Groundness analysis detects which program variables are bound to ground terms and is important in detecting determinacy, simplifying unification, etc. In [16] 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 [3] 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 δ. The resulting method is also simple to implement, and the analysis times based on a prototype analyser are promising at this initial stage.

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 annotated with program points @, δ, @, @ and @.

\[
\text{query}(x, y) :\begin{array}{l}
\text{δ} \text{delay}(x, y), \delta x = [ ] @. \\
\text{block delay}(-, ?). \\
delay(x, y) :\begin{array}{l}
\text{δ} x = y @. 
\end{array}
\end{array}
\]

The equations \(\phi_a = \phi_b = \varepsilon, \phi_c = \{ x = [ ], y = [ ] \}\) represent the actual bindings on \(x\) and \(y\) at @, δ and @ respectively.

Our method basically transforms the program into 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 concrete unification \(x = [ ]\) has been abstracted by the propositional formula \(gr(x)\). The block declaration for delay\((x, y)\) is modelled by the formula \(gr(x) \rightarrow ((gr(x) \leftrightarrow gr(x')) \land (gr(y) \leftrightarrow gr(y'))\). The concrete 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.

\[
\text{query}(x, y) :\begin{array}{l}
\text{δ} \text{delay}(x, y). @ \{ gr(x) \} @. \\
delay(x, y) :\begin{array}{l}
\text{δ} \{ gr(x) \rightarrow ((gr(x) \leftrightarrow gr(x')) \land (gr(y) \leftrightarrow gr(y')) \land (gr(x') \leftrightarrow gr(y'))\} \} @. 
\end{array}
\end{array}
\]

Restricting the formula on the right hand side of 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 \(gr(x) \land (gr(x) \leftrightarrow gr(y))\). This is logically equivalent to \(gr(x) \land \neg gr(y)\). 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\((x, y)\) 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 \(p(x)\) where \(p\) is a predicate symbol and \(x\) 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 \(h \leftarrow b\) where \(h\), the head, is an atom and \(b\), the body, is a finite sequence of literals. \(Prog\) is the set of programs, that is, the set of finite sets of clauses. \(Var, Term, Atom, Eqn, Lit, Goal\)

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

De®nition 3 call patterns.

For a given program unde®ned to re¯ect the scheduling behaviour of implementat ions like SICStus [1] and IF/Prolog [17].

such an atom will usually cause an error. These atoms are reduc ed, however, to avoid the need for an error state and

The operational semantics of a program is in terms of its quali®ed answers. Given a derivation from a state

it ¯ounders otherwise. Call patterns can now be de®ned in terms of the operational semantics.

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 substitutionθ to a set of variables U is denoted by θ | U and the composition of two substitutions θ and φ is denoted by θ ◦ φ and de®ned such that (θ ◦ φ)[u] = φ(θ(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 substitutions, that is, eqn(θ) = {u = t | u → t ∈ θ }, and mgu[E] denotes the set of most general uni®ers for an equation set E.

Let ≤ denote the instance ordering, that is, θ ≤ θ′ iff there exists a substitution σ ∈ Sub such that θ = σ ◦ θ′. Similarly, a syntactic object o is an instance of another o′, denoted o ≤ o′, iff there exists a substitution σ such that o = σ(o′). Instance lifts to φ(Atom) by I ≤ I′ iff for all a ∈ I there exists a′ ∈ I′ such that a ≤ a′. This, in turn, de®nes the equivalence relation ∼, that is I ∼ I′ iff I ≤ I′ and I′ ≤ I. Finally, the (quotiented) set of interpretations Int = φ(Atom)/ ∼ is ordered by [I] ∼ ≤ [I′] ∼ iff I ≤ 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 de®ned by State = God × Sub × God. A state (G, θ, D) records a sequence of literals G, the current substitutionθ, and a sequence of delayed atoms D. For a given state s and a program P, the relation ⊑s is de®ned such that h ← b ⊑s P iff there exists ρ ∈ Ren such that ρ(h ← b) ∈ P and var(h ← b) ∩ var(s) = ∅. Similar to [14], to abstract away from particular language considerations, the operational semantics is de®ned in terms of two parametric functions reduce and woken. (In [14], a delay truth function is used instead with delay ⇔ ¬reduce.) We say the atom a reduces with the substitution θ whenever reduce(a, θ) holds, whereas woken(D, θ) denotes the subsequence of atoms in D that are woken with the substitution θ. These functions are assumed to satisfy four conditions:

1. a ∈ woken(D, θ) iff a ∈ D and reduce(a, θ).
2. reduce(a, θ) iff reduce(ρ(a), ρ(θ)) for all ρ ∈ Ren.
3. reduce(a, θ) iff reduce(a, θ | var(s)).
4. if θ ≤ θ and reduce(a, θ) then reduce(a, θ).

**Definition 1 reduction.** The relation s→p s′ where s, s′ ∈ State is de®ned as follows:

- if s = (x = t :: G, θ, D) then s′ = (D :: G, θ′, D \ D′) where θ′ = mgu({x = t} ∪ eqn(θ)) and D′ = woken(D, θ′);
- if s = (a :: G, θ, D) and reduce(a, θ) holds then s′ = (b :: G, θ′, D) where h ← b ⊑s P and θ′ = mgu({h = a} ∪ eqn(θ));
- if s = (a :: G, θ, D) and reduce(a, θ) fails then s′ = (G, θ, a :: D).

Note that when an atom a has no de®ning clause, reduce(a, θ) 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 unde®ned to re®ect the scheduling behaviour of implementations like SICStus [1] and IF/Prolog [17].

**Definition 2 derivation.** A derivation from a state s for a program P is a (®nite or in®nite) sequence of reductions s1→p s2→p . . . where s = s1.

The operational semantics of a program is in terms of its quali®ed answers. Given a derivation from a state s and a program P with last state (θ, D) we say the tuple (θ, D) is a quali®ed answer to s. It is successful if D = ε and it flounders otherwise. Call patterns can now be de®ned in terms of the operational semantics.

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

\[ \text{call}(P, G) = \{\{l, \theta | \text{var}(l)\} | \langle G, \varepsilon, \varepsilon \rangle \rightarrow^p \langle l :: G', \theta, D \rangle\} \]
4 Abstract Domains

Abstract substitutions A Pos domain is used to capture positive information about downward-closed predicates. Following [5], we let \( \Omega_S(\{\land, \lor, \leftrightarrow, \neg\}) \) denote the set of propositional formulae formed from the set of connectives \( \{\land, \lor, \leftrightarrow, \neg\} \) and a set of propositional symbols \( S \). A truth assignment is a function \( r : S \rightarrow \{0, 1\} \). Given \( f, f' \in \Omega_S(\{\land, \lor, \leftrightarrow, \neg\}) \), \( r \models f \) denotes that \( r \) satisfies \( f \) and the notation \( f \models f' \) abbreviates \( r \models f' \). 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 \( \{gr(v), nv(v) \mid v \in V\} \). More generally, given \( n \) properties, \( p_1, \ldots, p_n \), the set of propositional symbols is \( \{p_i(v) \mid v \in V \land 1 \leq i \leq n\} \). As notation, we will usually add the \( v \) annotation to denote abstract substitutions, that is, to indicate propositional formulae.

**Definition 4.** \( Pos_V = \{f \in \Omega_S(\{\land, \lor, \leftrightarrow, \neg\}) \mid v \models f\} \) where \( S = \{p_i(v) \mid v \in V\} \) 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 [3, 5, 15]. \( Pos_V / \equiv \{\models\} \) is a complete lattice with lub \( \lor \) and glb \( \land \) and the least and greatest elements \( false \) and \( true \) [5].

Abstraction and concretisation of substitutions To formalise the relationship between a substitution and a formula an auxiliary function \( assign_\psi \) 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 \( \overline{f} \) is used to test for a property in the concrete whereas \( p_i \) is used for a symbol in abstract formulae.

**Definition 5.** \( assign_\psi(\phi) = \bigwedge_{i=1}^n \bigwedge_v p_i(v) \Leftrightarrow \overline{f_i}(\phi(v)) \)

Note that \( assign_\psi(\phi) \) is not necessarily positive.

**Example 1.** Suppose \( V = \{x, y, z\}, p_1 = gr, p_2 = nv, \phi = \{x \mapsto f(y)\} \) and \( \phi' = \{x \mapsto f(y), y \mapsto 1\} \) (note that \( \phi' \leq \phi \)). Then \( assign_\psi(\phi) = \neg gr(x) \land nv(x) \land \neg gr(y) \land \neg nv(y) \land \neg gr(z) \land \neg nv(z) \) and \( assign_\psi(\phi') = gr(x) \land nv(x) \land gr(y) \land nv(y) \land gr(z) \land nv(z) \).

We can now define the concretisation and abstraction functions \( \gamma \) and \( \alpha \).

**Definition 6.**

\[
\gamma_V : Pos_V \rightarrow \phi(Sub), \quad \alpha_V : \phi(Sub) \rightarrow Pos_V
\]

\[
\gamma_V(\phi') = \{\phi \mid \forall \phi \leq \phi \Rightarrow assign_\psi(\phi) \models \phi\}, \quad \alpha_V(\phi) = \bigwedge_{\phi \subseteq \gamma_V(\phi')} \phi
\]

Note that defining \( \gamma_V \) on \( Pos_V \) ensures that \( \alpha_V(\phi) \in Pos_V \).

**Example 2.** Suppose \( V = \{x, y, z\}, p_1 = gr, p_2 = nv \) and \( \phi = \{x \mapsto f(y)\} \). Let \( \phi' = (gr(x) \leftrightarrow gr(y)) \land nv(x) \). Then \( \phi \in \gamma_V(\phi') \) since \( assign_\psi(\phi') \models \phi' \) for all \( \phi' \leq \phi \).

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

**Proposition 7.** \( \langle \phi(Sub) \rangle (\subseteq), \alpha_V, Pos_V (\models), \gamma_V \) 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 8.**

\[
\text{Atom}^\dagger = \{\langle p(x), \phi^\dagger \rangle \mid p \in P_{red} \land \phi^\dagger \in Pos_{var(x)}\}
\]

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

**Definition 9.**

\[
\gamma : \phi(\text{Atom}^\dagger) \rightarrow Int
\]

\[
\gamma(\phi) = \{\langle p(x), \phi \mid var(x) \rangle \mid \langle p(x), \phi^\dagger \rangle \in \text{Int}^\dagger \land \phi \in \gamma_{\text{var}(x)}(\phi^\dagger)\}
\]
\( \gamma \) induces a natural equivalence relation on \( \varphi(\text{Atom}^1) \), that is \( I^1 \cong I^1' \) if \( \gamma(I^1) = \gamma(I^1') \). This in turn is used to define \( \text{Int}^1 = \varphi(\text{Atom}^1)/\cong \), the abstract analogue of \( \text{Int} \). An ordering for \( \text{Int}^1 \) is given by \( [I^1]_\cong \subseteq [I^1']_\cong \) iff \( \gamma([I^1]_\cong) \subseteq \gamma([I^1']_\cong) \) and the lub is defined by \( \cup_{i \in I^1} [I^i]_\cong = [\cup_{i \in I^1} I^i]_\cong \).

**Definition 10.**

\[
\gamma : \text{Int}^1 \rightarrow \text{Int} \quad \alpha : \text{Int} \rightarrow \text{Int}^1
\]

\[
\gamma(I^1) = [\gamma(I^1)]_\cong \quad \alpha(I) = \bigsqcup_{i \subseteq \gamma([I^1]_\cong)} [I^i]_\cong
\]

We can thus prove:

**Proposition 11.** \( \langle \text{Int}^1, (\subseteq), \text{Int} \rangle \) is a Galois connection.

## 5 Abstract Compilation

First we define a safety condition for the abstract reduce function which is represented by the propositional formula \( \text{reduce}^1(p(x), \phi^1) \). The intention is that an abstract goal will never reduce if its corresponding concrete goal never reduces. The safety condition to impose this is given below.

**Definition 12.** \( \text{reduce}^1(p(x), \phi^1) \rightarrow \forall \phi \in \gamma_{\text{var}(x)}(\phi^1). \text{reduce}(p(x), \phi) \).

**Example 3.** Let us define \( \text{reduce}^1(p(x), \phi^1) \leftrightarrow \forall \phi \in \gamma_{\text{var}(x)}(\phi^1). \text{reduce}(p(x), \phi) \). In SICStus Prolog, given the declaration \( \text{block} p(d_1, ..., d_n) \) for the \( n \)-ary predicate \( p \), where \( d_i \in \{-, ?\} \) for \( 1 \leq i \leq n \), then an abstract reduce function is:

\[
\text{reduce}^1(p(x), \phi^1) \leftrightarrow \forall \phi \mid \phi^1 \mid = \bigvee_{d_i = -} \text{nu}[x_i]
\]

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 13 abstract compilation \( \alpha \).**

\[
\alpha[c_1, ..., c_n] = \alpha_{\text{clause}}[c_1], \alpha_{\text{bot}}[c_1], ..., \alpha_{\text{clause}}[c_n], \alpha_{\text{bot}}[c_n]
\]

\[
\alpha_{\text{clause}}[p(x) \leftarrow b] = p(x) \leftarrow (\phi^1 \rightarrow \phi^1_b), \alpha_{\text{goal}}[b]
\]

\[
\alpha_{\text{bot}}[p(x) \leftarrow b] = p(x) \leftarrow \phi^1_b
\]

\[
\alpha_{\text{goal}}[I_1, ..., I_n] = \alpha_{\text{literal}}[I_1], ..., \alpha_{\text{literal}}[I_n]
\]

\[
\alpha_{\text{literal}}[p(x)] = p(x)
\]

\[
\alpha_{\text{literal}}[t = t'] = \phi^1_b
\]

where

\[
\phi^1_b = \bigwedge_{i \in x} \text{nu}[v]
\]

\[
\phi^1_b = \bigwedge_{\text{reduce}(p(x'), \phi^1)} \phi^1
\]

\[
\phi^1_b = \alpha_{\text{var}(x - x')}^{\text{mu}(x = x')}
\]

\[
\phi^1_b = \alpha_{\text{var}(t - t')}^{\text{mu}(t = t')}
\]

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

**Example 4.** The abstract program for the program \( \text{permute} \) listed in section 1 is below. First note we have written \( \text{new} \) to denote the predicate \( \text{new}_{\text{remove}} \). The first (normalised) clause for \( \text{permute} \) is

\( \text{permute}(u, v) \leftarrow u = [], v = [] \).

This is abstracted by the propositional formula \( \text{nu}[u] \land \text{gr}(u) \land \text{nu}[v] \land \text{gr}(v) \). When \( \alpha_{\text{bot}} \) is applied to this same clause the propositional formula \( \phi^1_b \) obtained is again \( \text{nu}[u] \land \text{gr}(u) \land \text{nu}[v] \land \text{gr}(v) \).

Next we turn to how the block declaration
Deﬁnition 15. If an abstract state \( s \) satisﬁes \( \varphi^1 \), then an abstract derivation from \( s \) is an (ﬁnite or inﬁnite) sequence of abstract reductions \( s^1 \to s_2^1 \to s_3^1 \to \ldots \) where \( s^1 \) is the initial state and \( s_i^1 \) is the state at step \( i \). Abstract call patterns can now be deﬁned in terms of the abstract semantics.

Deﬁnition 16. \( \text{call}^1(P_{G^1}) = [(l : \varphi^1 \mid \text{var}(l)) \mid \langle G^1, \text{true} \rangle \rightarrow 

The theorem below ensures the safety of the analysis. Due to space limitations, the proof is omitted.

Theorem 17. If \( P^1 = \alpha[P], \) and \( G^1 = \alpha_{\text{goal}}[G] \) then
\[
\text{call}(P_G) \leq \gamma(\text{relax}(\text{call}(P_{G^1})))
\]
6 Implementation

A prototype analyser has been implemented in SICStus Prolog which takes, as input, SICStus Prolog programs with block declarations and produces, as output, the call patterns for each predicate in the program. The analyser is largely based on [3] and essentially infers which program variables are bound to ground and non-variable terms. Following [3], Pos formulae are represented as their models and n-ary predicates iff(x, x1, ..., xn) are used to express formulae of the form x ↔ x1 ∧ ... ∧ xn⁻₁.

Example 5. Consider again the abstract permute program and, in particular, the new and reduce predicates of example 4. In the case of new, each of its clauses are translated into a clause by expressing Pos formulae, like \( \text{n}_v(x') \) and \( \text{gr}(x') ↔ (\text{gr}(v') \land \text{gr}(y)) \), in terms of iff predicates, like iff(nv(x')) and iff(gr(u'), gr(v'), gr(y)). The single reduce clause, on the other hand, is expanded into three clauses, the formulae \((x_1 \lor x_2) \rightarrow x_3\) being logically equivalent to \((x_1 \land x_3) \lor (x_2 \land x_3) \lor (x_1 \land \lnot x_2)\). The neg(x) predicate expresses \(\lnot x\). Note that although \(\lnot x\) 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.

\[
\begin{align*}
\text{new}(u, v, w, x) :&\, \text{new}(u, v, w, x), \text{new}(u', v', w', x'), \\
\text{reduce}(\text{new}(u, v, w, x), \text{new}(u', v', w', x')) :&\, \text{iff}(n_v(u)), \\
\text{iff}(n_v(u'), \text{iff}(\text{gr}(u'), \text{gr}(v'))), \\
\text{iff}(n_v(w'), \text{iff}(\text{gr}(w'), \text{gr}(y))).
\end{align*}
\]

Finally, a magic transform (see, e.g. [3]) coupled with bottom-up evaluation is used to calculate the call patterns of the predicates.

Analysers for Pos which capture call patterns through models and magic can be both fast and efficient [3]. The main purpose of the implementation was to verify that the analysis could indeed accurately trace synchronisation behaviour. To this end we analysed five small programs that involved high degrees of coroutining and compared the results produced to those obtained by handworking (!) the suspension-freeness analysis of [6] for the \(n\) and \(\text{gr} \) Pos domain. We also compared our analysis to a naive transform that abstracted possibly delaying predicates by true. This transformation is safe because Pos is monotonic. The programs analysed were: money.pl, a cryptarithmic problem; permute.pl, the naive sort listed in the introduction; primes.pl, a lazy primes sieve; queens.pl, a coroutining \(n\)-queens; and monotonic.pl, a monotonic permutation generator.

<table>
<thead>
<tr>
<th>program</th>
<th>clauses</th>
<th>time (secs)</th>
<th>precision (modes)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>naive delay</td>
<td>naive susp</td>
</tr>
<tr>
<td>money.pl</td>
<td>6</td>
<td>21.96</td>
<td>9.13 same</td>
</tr>
<tr>
<td>permute.pl</td>
<td>9</td>
<td>0.05</td>
<td>0.15 better</td>
</tr>
<tr>
<td>primes.pl</td>
<td>12</td>
<td>0.07</td>
<td>0.07 same</td>
</tr>
<tr>
<td>queens.pl</td>
<td>12</td>
<td>0.34</td>
<td>0.36 same</td>
</tr>
<tr>
<td>monotonic.pl</td>
<td>21</td>
<td>0.19</td>
<td>20.94 better</td>
</tr>
</tbody>
</table>

The times given are in seconds and have been taken on a Sun Sparc10 machine. The clauses column of the table is the number of clauses in the program. The naive and delay columns give the analysis times for the naive transform and the transform described in this paper. Standard techniques, like using SCCs, have not yet been incorporated into the analyser [3]. The naive and susp precision columns compare our analysis results to those generated by the naive analysis and the suspension-freeness analysis. Interestingly, our analysis either outperforms the naive analysis on speed or precision (see money.pl, permute.pl and monotonic.pl) or performs similarly. When comparing with the suspension-freeness analysis, precision is improved for permute.pl and monotonic.pl but not for money.pl.

money.pl
is slow because some clauses involve large numbers of variables. To conclude, the analysis looks promising and we are currently completing an analyser for \( \text{klic} \) programs so that we can verify that the analysis scales to large, more complex programs.

7  Related Work

Suspension-analysis framework The simple and practical framework of [6], 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 [6]) the analysis cannot infer modes to the precision of our framework.

Multiset framework The multiset framework of [14] uses a multiset to record which atoms are definitely delayed and which atoms are possibly delayed. However, in [10] (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 [10]. The semantics improves the precision but efficient implementation is still difficult. For example, for the approach to be practical, the implementation described in [10] makes “observational equivalence” of closures, represents closures in a special way, and uses a number of techniques, including the differential approach [9], to improve the handling of cases when no atoms are delayed. Futhermore, 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 [16], 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 [18] which removes synchronisation from a concurrent constraint program. The work of [18] 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 [10] with the abstract compilation work of [3] and [7]. Abstract compilation and abstract programs have been proposed for deriving modes [3, 4], encoding regular approximations [8], inferring directional types [2] and deducing inter-argument relationships [11]. In addition, [3, 13, 11] and [4] 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 monotonic domains and then later to non-monotonic domains. A predictable wakeup order for delayed goals would certainly enable precision to be improved and may even open up new possibilities of efficient analyses for non-monotonic domains. Another direction for future work will be to investigate how demand-driven analysis can be used to infer the modes that suspending goals are reawoken with. On the implementation side, the klic analyser will soon be complete.

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.

References


17. A. Verden. Personal communication on freeze/2 and the const\_delay module of IF/Prolog V5.0. April 1996.