Bottum-up abstract interpretation of logic programs

Eindhoven University of Technology

Department of Mathematics and Computing Science

Bottum-up Abstract Interpretation of Logic Programs

by

M. Codish D. Dams E. Yardeni

92/24

Computing Science Note 92/24
Eindhoven, September 1992

COMPUTING SCIENCE NOTES

This is a series of notes of the Computing Science Section of the Department of Mathematics and Computing Science Eindhoven University of Technology.
Since many of these notes are preliminary versions or may be published elsewhere, they have a limited distribution only and are not for review.
Copies of these notes are available from the author.

Copies can be ordered from:
Mrs. F. van Neerven
Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
ISSN 0926-4515

editors: prof.dr.M.Rem
prof.dr.K.M.van Hee.

Bottom-up Abstract Interpretation of Logic Programs*

Michael Codish[‡] Dennis Dams[§] Eyal Yardeni[¶]

Abstract

This paper presents a formal framework for the bottom-up abstract interpretation of logic programs which can be applied to approximate *answer substitutions*, *partial answer substitutions* and *call patterns* for a given program and arbitrary initial goal. The framework is based on a T_P like semantics defined over a Herbrand universe with variables which has previously been shown to determine the answer substitutions for arbitrary initial goals. The first part of the paper reconstructs this semantics to provide a more adequate basis for abstract interpretation. A notion of *abstract substitution* is introduced and shown to determine an abstract semantic function which for a given program can be applied to approximate the answer substitutions for an arbitrary initial goal.

The second part of the paper extends the bottom-up approach to provide approximations of both partial answer substitutions and call patterns. This is achieved by applying *Magic Sets* and other existing techniques to transform a program in such a way that the answer substitutions of the transformed program correspond to the partial answer substitutions and call patterns of the original program. This facilitates the analysis of concurrent logic programs (ignoring synchronization) and provides a collecting semantics which characterizes both success and call patterns.

*To appear in *Theoretical Computer Science*.

[‡]Department of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A, B-3030 Heverlee, Belgium

[§]Department of Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, the Netherlands

[¶]Department of Computer Science, Weizmann Institute of Science, Rehovot 76100, Israel

1 Introduction

The framework of abstract interpretation provides the basis for a semantic approach to dataflow analysis. A program analysis is viewed as a non-standard semantics defined over a domain of data descriptions where the syntactic constructs in the program are given corresponding non-standard interpretations. For a given language, different choices of a semantic basis for abstract interpretation may lead to different approaches to analysis of programs in that language. For logic programs we distinguish between two main approaches: "bottom-up analysis" and "top-down analysis" [18]. The first is based on a bottom-up semantics such as the standard T_P semantics, the latter on a top-down semantics such as the SLD semantics.

The meaning of a logic program P in the standard T_P semantics is the set of ground atoms in P's vocabulary which are implied by the program. An abstraction of the T_P function will typically provide an approximation of a program's (ground) *success patterns* and hence provide the basis for applications such as type analysis [15, 33]. In a top-down semantics, the meaning of a program usually associates with an initial goal the set of answer substitutions for that goal. The semantics is usually based on some form of SLD resolution possibly considering a specific computation rule. Typically, a top-down semantics is extended to a *collecting semantics* in which the *call patterns* of a program are recorded. The call patterns for a program (with an initial goal) specify the set of calls that may arise in computations. Such information determines how each clause in the program might be called and hence provides the basis for program specialization. A typical example is mode analysis (e.g., [20]).

Abstractions of the T_P semantics are often not useful for program analysis as they describe only ground instances of atoms. In addition, they are not useful for analysing concurrent logic programs as they describe only success patterns while concurrent programs are also characterized by failing and diverging computations. Furthermore, as the evaluation of a bottom-up semantics does not correspond to the operational behaviour of a program (which is top-down in nature), it is not readily extended to provide information about call patterns.

This paper presents a framework for the bottom-up abstract interpretation of logic programs which attempts to overcome these deficiencies. The proposed framework provides a uniform approach to the analysis of logic programs which is shown suitable to approximate success patterns, partial success patterns and call patterns. The concrete semantics which is the foundation of our framework is based on the work of Falaschi *et al.* [13] which presents a bottom-up semantics for logic programs defined over a domain of non-ground Herbrand interpretations. The advantage of this semantics is that it captures the operational notion of the logic variable. The meaning of a program is a set of non-ground atoms which is shown to determine the set of answer substitutions for an arbitrary initial goal. This semantics provides the basis for the bottom-up analysis of logic programs as described in [3, 4, 8, 16]. In this paper we reconstruct the semantics defined in [13] to provide a more natural basis for abstract interpretations which are determined by a notion of abstract substitutions. The resulting framework is similar to those defined independently in [3, 4] and [16] and can be applied to approximate the answer substitutions for a given program and arbitrary initial goal.

The main contribution of this paper is in the extensions of this framework. The analysis of concurrent logic programs is facilitated by extending the framework to approximate the *partial answer substitutions* for a program with an initial goal. Partial answer substitutions

are substitutions which correspond to the results of partial computations of the initial goal, regardless if these computations eventually succeed, fail or diverge. This extension turns out to be especially simple given the result of Falaschi and Levi [12] which demonstrates how to augment a program by adding to it a set of unit clauses so that the answer substitutions of the augmented program are precisely the partial answer substitutions of the original program. This implies that the partial answer substitutions of the original program can be approximated by approximating the answer substitutions of the augmented program.

A similar approach is applied to approximate the call patterns of a logic program. We use the Magic Set method [5, 7] to transform a program with initial goal so that the answer substitutions of the transformed program correspond to the call patterns of the original program. We demonstrate this technique for both sequential and concurrent logic programs and prove its correctness.

The rest of the paper is organized as follows. Section 2 gives the notation and preliminary definitions which will be used throughout. Section 3 reconstructs the bottom-up semantics originally defined in [13] to provide the foundation for our bottom-up framework. Section 4 presents a general framework for abstractions which are determined by a given definition of abstract substitutions. Section 5 notes that analyses based on the definitions in the previous section are inherently exponential. A *join* operator is defined and used to define a more abstract semantic definition. Section 6 describes extensions of the framework to provide approximations of partial answer substitutions and call patterns. Finally, Section 7 concludes.

2 Preliminaries

In the following we assume familiarity with the standard definitions and notation for logic programs [17]. Throughout, Σ, Π and *Var* will respectively denote a set of function symbols, a set of predicate symbols and a denumerable set of variables. The non-ground term algebra over Σ and *Var* is denoted $Term(\Sigma, Var)$ or *Term* for short. The set of atoms constructed from predicate symbols in Π and terms from *Term* is denoted $Atom(\Pi, \Sigma, Var)$ or *Atom* for short. Goals are finite sequences of atoms. A goal is typically denoted by $\bar{a}$, by $\langle a_1, \ldots, a_n \rangle$ or simply by $a_1, \ldots, a_n$. The empty atom sequence is denoted by *true*. A logic program is a finite set of Horn clauses of the form $h \leftarrow \bar{b}$ where h is an atom, called the *head*, and $\bar{b}$ is a goal, called the *body*. The sets of atoms which occur as heads and in bodies of the clauses of a logic program P are denoted $heads(P)$ and $atoms(P)$ respectively. We often write P_G for a program P with initial goal G. In this case, $atoms(P_G)$ includes the atoms of G.

Substitutions. A *substitution* is a mapping from *Var* to *Term* which acts as the identity almost everywhere. It extends to apply to any syntactic object in the usual way. A substitution θ is finitely represented by the set $\{x \mapsto \theta(x) \mid \theta(x) \neq x\}$. The identity substitution is denoted by ϵ. The application of a substitution θ to a syntactic object S is denoted by $S \cdot \theta$ (or $S\theta$) instead of $\theta(S)$. If Θ is a set of substitutions, then $S\Theta = \{S\theta \mid \theta \in \Theta\}$. Composition of substitutions ρ and σ and restriction of σ to $V \subseteq Var$ are defined as usual and denoted $\rho\sigma$ and $\sigma \restriction V$ respectively. A substitution σ is *idempotent* if $\sigma\sigma = \sigma$. In this paper we restrict our interest to idempotent substitutions, unless explicitly stated otherwise. The set of idempotent substitutions is denoted *Sub*. Note that *Sub* is not closed under composition. However, in the following, compositions are performed only when the result is guaranteed to be idempotent. We fix a partial function *mgu* which maps a pair of syntactic objects to an idempotent most

general unifier of the objects, if such exists. Thus, a statement $\theta = mgu(s,t)$ implies that s and t are unifiable.

Equivalence relations. If $\sim$ is an equivalence relation on a set X, we denote by $[x]_\sim$ the equivalence class of $x \in X$. When clear from the context we abbreviate $[x]_\sim$ by $[x]$. In the following we will often abuse notation and let the elements of a set denote their corresponding equivalence classes.

Renaming. A *variable renaming* is a (not necessarily idempotent) substitution which is a bijection on Var. Syntactic objects (e.g., atoms, sets of atoms) t_1 and t_2 are *equivalent up to renaming*, denoted $t_1 \sim t_2$, if, for some variable renaming ρ, $t_1\rho = t_2$. The set of variables that occur in a syntactic object t is denoted $vars(t)$. Given an equivalence class $\hat{t}$ of syntactic objects and a finite set of variables $V \subseteq Var$, it is always possible to find a representative t of $\hat{t}$ (i.e., an object t such that $[t] = \hat{t}$) which contains no variables from V.

For any syntactic object s, $\Upsilon_s : Atom/_\sim{}^* \to Atom^*$ is a function which takes a sequence of equivalence classes of atoms and returns a corresponding sequence of representatives which are renamed apart from the variables in s and from each other: $\Upsilon_s\langle \hat{a}_1, \ldots, \hat{a}_n\rangle = \langle a_1, \ldots, a_n\rangle$ such that for $1 \leq i,j \leq n$ and $i \neq j$: $[a_i] = \hat{a}_i$, $vars(a_i) \cap vars(s) = \emptyset$ and $vars(a_i) \cap vars(a_j) = \emptyset$. In the following we let Υ denote Υ_P where P is an implicitly assumed program.

Operational semantics. The operational semantics of a logic program is typically defined in terms of a *transition system* on $States = Atom^* \times Sub$. A program P is associated with the transition system $(States, \to_P)$, where: $\to_P \subseteq States \times States$ is the smallest relation such that $s \to_P s'$ if $s = \langle A_1, \ldots, A_i, \ldots, A_n; \theta\rangle$, $s' = \langle A_1, \ldots, A_{i-1}, B_1, \ldots, B_m, A_{i+1}, \ldots, A_n; \theta\sigma\rangle$, $H \leftarrow B_1, \ldots, B_m$ is a renaming of a clause from P (which contains no variables from s), and $\sigma = mgu(A_i\theta, H)$. The reflexive and transitive closure of $\to_P$ is denoted by $\to_P^*$; the subscript P will often be omitted when no confusion can arise. Given a program P, $\theta \in Sub$ is an *answer substitution* (or *answer* for short) for a goal G iff there is a substitution θ' such that $\langle G; \epsilon\rangle \to^* \langle true; \theta'\rangle$ and $\theta = \theta' \restriction vars(G)$.

3 Concrete semantics

This section presents a bottom-up semantics for logic programs which provides the basis for bottom-up abstractions in the following sections. It reconstructs the semantics defined in [13] to associate each clause head in a program with a set of substitutions. The meaning of a program then specifies for each clause head a set of its instances which are implied by the program. Conceptually, we would like to associate each clause in a program with a set of substitutions (representing instances of its head). It is however notationally more convenient to define the semantics as mapping *atoms* to sets of substitutions. In most cases, we can assume without loss of generality that a program's clauses are uniquely determined by its heads (e.g., by renaming clauses apart). However, the correctness of our formalization does not depend on this assumption.

Our decision to alter the semantics of [13] is motivated by the observation that abstractions for logic programs are naturally defined in terms of some notion of *abstract substitution*. That is, an abstract semantic domain is determined given a definition which specifies how to approximate (sets of) substitutions. It is straightforward to show that our semantics is consistent with that of [13] and hence can be applied to determine a mapping which associates an arbitrary goal with the set of its answer substitutions.

3.1 Concrete domain

The semantics will be defined in terms of mappings from atoms to sets of substitutions which are intended to specify instances of the heads of a program's clauses. Such mappings are lifted to an appropriate notion of equivalence up to renaming.

DEFINITION **3.1.** *concrete semantic domain*
We equip $(Atom \rightarrow 2^{Sub})$ *with the preordering* $\preceq$ *defined by*

$$f_1 \preceq f_2 \;\Leftrightarrow\; \forall_{a \in Atom}\ [a \cdot f_1(a)] \subseteq [a \cdot f_2(a)].$$

The equivalence relation induced by $\preceq$ *is denoted* $\sim_{Int}$ *and the corresponding partial order by* $\sqsubseteq_{Int}$. *The concrete domain of* interpretations *is defined by* $Int = (Atom \rightarrow 2^{Sub})/_{\sim_{Int}}$.

It follows that:

PROPOSITION **3.2.**
$(Int, \sqsubseteq_{Int})$ *is a complete partial order with bottom element* $\bot_{Int} = \lambda_{a \in Atom}\ .\ \emptyset$.

In the following we will refer to the *atoms* of an interpretation:

DEFINITION **3.3.** *atoms of an interpretation*

$$atoms : Int \rightarrow 2^{Atom/\sim}$$
$$atoms(f) = \left\{\ [a\theta] \;\middle|\; a \in Atom,\ \theta \in f(a)\ \right\}.$$

3.2 Concrete semantics

Falaschi *et al.* [13] define a semantic operator[1] $S_P : 2^{Atom/\sim} \rightarrow 2^{Atom/\sim}$ similar to the standard T_P operator. The semantics of a program P is the least fixed point of S_P which specifies a set of non-ground atoms which are implied by P. This semantics is attractive for purposes of program analysis as it captures both declarative and operational aspects of programs: *operational* because the meaning of a program is shown to determine the answer substitutions for arbitrary goals, *declarative* because the set of ground instances of a program's meaning corresponds to the standard minimal model (T_P) semantics.

In the following definitions we reformulate this semantics of [13]. The meaning of a program P is defined as the least fixed point of an operator $F_P : Int \rightarrow Int$ which maps each clause head to a set of substitutions. In the sequel it will be convenient to lift the *mgu* function as follows:

DEFINITION **3.4.** *lifted mgu*
Let P *be a program and* $f \in Int$. *Define the function* $mgu_f : Atom^* \times Atom^* \rightarrow 2^{Sub}$ *as follows:*

$$mgu_f(\bar{b}, \langle a_1, \ldots, a_n \rangle) = \left\{\ mgu(\bar{b}, \Upsilon_P\langle a_1\theta_1, \ldots, a_n\theta_n \rangle) \;\middle|\; \begin{array}{l} 1 \le i \le n, \\ \theta_i \in f(a_i) \end{array}\ \right\}$$

[1] This semantics is called the "S semantics" in [13] where the operator is denoted T_S.

Note that for any $f \in Int$, $mgu_f(\langle\rangle, \langle\rangle) = \{\epsilon\}$. Furthermore, observe that a unification of the form $mgu_f(\bar{b}, \bar{b})$ has the effect of "combining" the substitutions which f associates with the respective atoms in $\bar{b}$.

DEFINITION **3.5.** *concrete semantic function*
Let P be a logic program. Define $F_P : Int \to Int$ by:

$$F_P(f) = \lambda_h \, . \bigcup \left\{ \Theta \;\middle|\; \begin{array}{l} h \leftarrow \bar{b},\ \bar{a} \in Atom^*, \\ \Theta = mgu_f(\bar{b}, \bar{a}) \end{array} \right\}$$

Note that F_P is well-defined because $f \sim f'$ implies that $h \cdot mgu_f(\bar{b}, \bar{a}) \sim h \cdot mgu_{f'}(\bar{b}, \bar{a})$.

PROPOSITION **3.6.**
The function $F_P : Int \to Int$ is monotonic and continuous.

PROOF. Standard. □

We denote the concrete meaning of a program P by $[\![P]\!]_{con} = lfp(F_P)$. It is straightforward to show that $[\![P]\!]_{con}$ is consistent with the S semantics defined in [13] in the sense that $atoms([\![P]\!]_{con}) = lfp(S_P)$.

3.3 Answer substitutions

To show how the meaning of a program determines the answer substitutions for arbitrary initial goals, we follow [13] and introduce the following:

DEFINITION **3.7.** *answers determined by an interpretation*
Let $f \in Int$. Define $ans_f : Atom^* \to 2^{Sub}$ by

$$ans_f(\bar{b}) = \bigcup \left\{ \; mgu_f(\bar{b}, \bar{a}) \restriction vars(\bar{b}) \;\middle|\; \bar{a} \in Atom^* \; \right\}$$

LEMMA **3.8.**
Let P be a logic program. Then $\theta \in Sub$ is an answer for a goal G iff there exists $\theta' \in ans_{[\![P]\!]_{con}}(G)$ such that $G\theta \sim G\theta'$.

PROOF. Follows directly from the observation that $atoms([\![P]\!]_{con}) = lfp(S_P)$ and the strong soundness and completeness results of [14]. □

EXAMPLE **1.** *Let P be the program*[2]*:*

```
member(x,[x|xs]).
member(x,[_|xs]) :- member(x,xs).
```

[2] A "_" denotes an anonymous variable.

$[\![P]\!]_{con}$ maps the first clause (head) to the set $\{\epsilon\}$ (taking $n = 0$ in Definition 3.5); it maps the second clause (head) to the set

$$\{\{xs \mapsto [x \mid \_]\}, \{xs \mapsto [\_, x \mid \_]\}, \{xs \mapsto [\_, \_, x \mid \_]\}, \ldots\}.$$

The function $ans_{[\![P]\!]_{con}}$ maps the goal member(1,[2,3]) to $\emptyset$; it maps the goal member(1,[1,2,3]) to $\{\epsilon\}$ and it maps the goal member(x,[1,2,3]) to the set

$$\{\{x \mapsto 1\}, \{x \mapsto 2\}, \{x \mapsto 3\}\}.$$

4 Abstract Semantics

This section defines a framework for the bottom-up abstract interpretation of logic programs. A particular abstraction is determined by specifying how sets of substitutions are to be abstracted. An appropriate notion of abstract substitution is then shown to determine an abstract semantic domain and an abstract semantic function. This follows the spirit of the denotational approach to abstract interpretation defined by Nielson [23] and advocated by Marriott and Søndergaard (e.g., [19]). The abstract meaning of a program is a mapping which associates an abstract substitution with each clause head. This mapping is shown to approximate the concrete semantics from the previous section and can hence be used to approximate the set of answer substitutions for an arbitrary initial goal.

We assume the standard framework of abstract interpretation as defined in [10] in terms of *Galois insertions*.

DEFINITION **4.1.** *Galois insertion [21]*
A Galois insertion is a quadruple (E, α, D, γ) where:

1. *$(E, \sqsubseteq_E)$ and $(D, \sqsubseteq_D)$ are complete lattices called* concrete *and* abstract domains *respectively;*
2. *$\alpha : E \to D$ and $\gamma : D \to E$ are monotonic functions called* abstraction *and* concretization functions *respectively; and*
3. *$\alpha(\gamma(d)) = d$ and $e \sqsubseteq_E \gamma(\alpha(e))$ for every $d \in D$ and $e \in E$.*

We say that elements of D describe *elements of E. By abuse of notation, we sometimes let D denote both the abstract domain and the Galois insertion. When $E = 2^{Sub}$ or $E = Int$ we call D a domain of* abstract substitutions *or* abstract interpretations *respectively.*

4.1 Abstract substitutions

In the following we construct a domain of abstract interpretations by associating abstract substitutions with the clause heads of a program. The intention is that an abstract substitution should describe instances of the head of the clause it is associated with. As different sets of substitutions Θ_1 and Θ_2 may denote equivalent instances of an atom a (namely, when $a{\cdot}\Theta_1 \sim a{\cdot}\Theta_2$), we impose an additional constraint on a domain of abstract substitutions, the purpose of which will become clear in Definition 4.3.

DEFINITION **4.2.** *abstraction substitutions*
A domain of abstract substitutions is a Galois insertion $(2^{Sub}, \bar{\alpha}, ASub, \bar{\gamma})$ *such that for every* $\Theta_1, \Theta_2 \in 2^{Sub}$ *and* $a \in Atom$,

$$a \cdot \Theta_1 \sim a \cdot \Theta_2 \;\Rightarrow\; a \cdot \bar{\gamma}\bar{\alpha}(\Theta_1) \sim a \cdot \bar{\gamma}\bar{\alpha}(\Theta_2).$$

Consider the following examples of domains of abstract substitutions.

EXAMPLE **2.** *identity*
It is straightforward to show that taking $(ASub, \sqsubseteq_{ASub}) = (2^{Sub}, \subseteq)$ *and* $\bar{\alpha} = \bar{\gamma} = id$ *provides a domain of abstract substitutions.*

EXAMPLE **3.** *dependency relations [9]*
A relation R *over a lattice* X *is* additive *if* $(x\ R\ x' \wedge y\ R\ y') \Rightarrow (x \sqcup y)\ R\ (x' \sqcup y')$. *A* dependency relation *is an additive equivalence relation (reflexive, symmetric and transitive) over* 2^{Var}. *We let* $dep(R)$ *denote the smallest dependency relation which contains a relation* R. *We say that a relation* R implies *a relation* R' *if* $dep(R) \supseteq dep(R')$. *We let* Dep *denote the complete lattice of relations over* 2^{Var} *modulo the equivalence induced by* dep *(i.e.,* $R \sim R' \Leftrightarrow dep(R) = dep(R')$*) ordered by implication. We let* $[W_1 \leftrightarrow W_1', \ldots, W_n \leftrightarrow W_n']$ *denote the relation* $\{(W_1, W_1'), \ldots, (W_n, W_n')\}$; *when sets are singleton, set brackets are dropped. It is straightforward to show that* $(2^{Sub}, \bar{\alpha}, Dep, \bar{\gamma})$ *is a domain of abstract substitutions where* $\bar{\alpha} : 2^{Sub} \to Dep$ *and* $\bar{\gamma} : Dep \to 2^{Sub}$ *are defined by:*

$$\bar{\alpha}(\Theta) = \left\{ (V, W) \;\middle|\; \forall_{\theta \in \Theta}\ vars(V\theta) = vars(W\theta) \right\}, \text{ and}$$

$$\bar{\gamma}(R) = \left\{ \theta \;\middle|\; \forall_{(V,W) \in R}\ vars(V\theta) = vars(W\theta) \right\}.$$

A dependency relation R *is intended to describe those substitutions* θ *which satisfy the condition that for every* $(V, W) \in R$ *(and hence in* $dep(R)$*), the terms in* $V\theta$ *are ground iff the terms in* $W\theta$ *are ground. A particular case is when* $V = \emptyset$ *(or respectively* $W = \emptyset$*); in this case it means that the terms in* $W\theta$ *(or respectively* $V\theta$*) are ground for any* θ.
For instance $\bar{\alpha}(\{x \mapsto f(a)\}) = [x \leftrightarrow \emptyset]$, $\bar{\alpha}(\{x \mapsto f(y)\}) = [x \leftrightarrow y]$, and $\bar{\alpha}(\{\epsilon\}) = \emptyset$.

EXAMPLE **4.** *function symbols*
Let $Sym = Var \to 2^{\Sigma}$ *be the complete lattice ordered by* $s_1 \sqsubseteq_{Sym} s_2 \Leftrightarrow \forall_x\ s_1(x) \subseteq s_2(x)$. *We denote by* $symbols(t)$ *the set of function symbols (and constants) occurring in a term (or set of terms)* t. *Let* $\bar{\alpha} : 2^{Sub} \to Sym$ *and* $\bar{\gamma} : Sym \to 2^{Sub}$ *be defined by*

$$\bar{\alpha}(\Theta) = \lambda_x\ .\ symbols(x\Theta), \text{ and}$$

$$\bar{\gamma}(s) = \left\{ \theta \;\middle|\; \forall_x\ symbols(x\theta) \subseteq s(x) \right\}.$$

For instance $\bar{\alpha}(\{x \mapsto f(a)\}) = \lambda_y$ *. if* $y \equiv x$ *then* $\{f, a\}$ *else* $\emptyset$, $\bar{\alpha}(\{x \mapsto f(y)\}) = \lambda_y$ *. if* $y \equiv x$ *then* $\{f\}$ *else* $\emptyset$, *and* $\bar{\alpha}(\{\epsilon\}) = \lambda_y\ .\ \emptyset$.
It is straightforward to show that $(2^{Sub}, \bar{\alpha}, Sym, \bar{\gamma})$ *is a domain of abstract substitutions.*

4.2 Abstract interpretations

A domain of abstract substitutions naturally lifts to a domain of abstract interpretations:

DEFINITION 4.3. *abstract interpretations*
Let $(2^{Sub}, \bar{\alpha}, ASub, \bar{\gamma})$ *be a domain of abstract substitutions. The* induced domain of abstract interpretations $(Int, \alpha, AInt, \gamma)$ *is constructed as follows:*

1. *The function* $\gamma : (Atom \to ASub) \to Int$ *is defined by:*

$$\gamma(g) = [\bar{\gamma} \circ g]_{\sim_{Int}}$$

2. *The domain* $Atom \to ASub$ *is equipped with the preordering* $\preceq$ *defined by:*

$$g_1 \preceq g_2 \;\Leftrightarrow\; \gamma(g_1) \sqsubseteq_{Int} \gamma(g_2)$$

3. *The equivalence relation induced by* $\preceq$ *is denoted* $\sim_{AInt}$ *and the corresponding partial order on* $AInt = (Atom \to ASub)/_{\sim_{AInt}}$ *is denoted* $\sqsubseteq_{AInt}$.

4. *The function* γ *is lifted to* $AInt \to Int$ *(by taking* $\gamma([g]_{\sim_{AInt}}) = \gamma(g)$*).*

5. *The function* $\alpha : Int \to AInt$ *is defined by:*

$$\alpha(f) = [\bar{\alpha} \circ f]_{\sim_{AInt}}$$

In the sequel we often introduce a domain of abstract interpretations AInt induced from a domain of abstract substitutions ASub and refer to the implicitly defined $\alpha, \gamma, \bar{\alpha}$ *and* $\bar{\gamma}$.

EXAMPLE 5. *Let AInt be the domain of abstract interpretations induced from* $ASub = Dep$ *and consider* $g_1, g_2 : Atom \to Dep$ *which map the atom* $p(x, y)$ *to* $[\{x, y\} \leftrightarrow \emptyset]$ *and* $[\{x, y, z\} \leftrightarrow \emptyset]$ *respectively (and map all other atoms to* $\bot_{Dep}$*). Observe that* $\gamma(g_1) = \gamma(g_2)$*; both specify the set of all ground instances of* $p(x, y)$*. This illustrates two points:*

1. $\gamma : (Atom \to ASub) \to Int$ *is not injective (and hence it is lifted to a function of type* $AInt \to Int$ *which is);*

2. *the equivalence relation* $\sim_{AInt}$ *on* $Atom \to ASub$ *has the effect of "restricting" elements of ASub to the variables in the corresponding atoms.*

LEMMA 4.4. *Let AInt be induced from ASub. Then* α *is well-defined.*

PROOF.

$$\begin{array}{lll}
 & f_1 \sim_{Int} f_2 & \\
\Leftrightarrow & \forall_h\ h \cdot f_1(h) \sim h \cdot f_2(h) & [\ \text{by the def. of } \sim_{Int}\ ] \\
\Rightarrow & \forall_h\ h \cdot \bar{\gamma}\bar{\alpha}(f_1(h)) \sim h \cdot \bar{\gamma}\bar{\alpha}(f_2(h)) & [\ \text{by Def. 4.2}\ ] \\
\Leftrightarrow & [\bar{\gamma} \circ \bar{\alpha} \circ f_1]_{\sim_{Int}} = [\bar{\gamma} \circ \bar{\alpha} \circ f_2]_{\sim_{Int}} & [\ \text{by the def. of } \sim_{Int}\ ] \\
\Leftrightarrow & \gamma(\bar{\alpha} \circ f_1) = \gamma(\bar{\alpha} \circ f_2) & [\ \text{by the def. of } \gamma\ ] \\
\Leftrightarrow & \bar{\alpha} \circ f_1 \sim_{AInt} \bar{\alpha} \circ f_2 & [\ \text{follows directly from Def. 4.3 (3)}\ ] \\
\Rightarrow & \alpha(f_1) = \alpha(f_2) & [\ \text{by the def. of } \alpha\ ]
\end{array}$$

□

LEMMA 4.5.
Let AInt be induced from ASub. Then $(Int, \alpha, AInt, \gamma)$ *is a Galois insertion.*

PROOF.

1. $(AInt, \sqsubseteq_{AInt})$ is a complete lattice with bottom element $\bot_{AInt} = \lambda_{a \in Atom} \cdot \bot_{ASub}$.

2. α is monotonic:

$$
\begin{array}{lll}
& f_1 \sqsubseteq_{Int} f_2 & \\
\Leftrightarrow & \forall_h\ [h \cdot f_1(h)] \subseteq [h \cdot f_2(h)] & [\ \text{by the def. of } \sqsubseteq_{Int}\ ] \\
\Rightarrow & \forall_h\ [h \cdot \bar{\gamma}\bar{\alpha}(f_1(h))] \subseteq [h \cdot \bar{\gamma}\bar{\alpha}(f_2(h))] & [\ \text{from Def. 4.2 (see Lemma 1 in apdx.)}\ ] \\
\Leftrightarrow & [\bar{\gamma} \circ \bar{\alpha} \circ f_1] \sqsubseteq_{Int} [\bar{\gamma} \circ \bar{\alpha} \circ f_2] & [\ \text{by the def. of } \sqsubseteq_{Int}\ ] \\
\Leftrightarrow & \gamma(\bar{\alpha} \circ f_1) \sqsubseteq_{Int} \gamma(\bar{\alpha} \circ f_2) & [\ \text{by the def. of } \gamma\ ] \\
\Leftrightarrow & [\bar{\alpha} \circ f_1] \sqsubseteq_{AInt} [\bar{\alpha} \circ f_2] & [\ \text{by the def. of } \sqsubseteq_{AInt}\ ] \\
\Leftrightarrow & \alpha(f_1) \sqsubseteq_{AInt} \alpha(f_2) & [\ \text{by the def. of } \alpha\ ]
\end{array}
$$

3. The monotonicity of γ follows directly from Definition 4.3.

4.

$$
\begin{array}{lll}
& \gamma\alpha(f) \sqsupseteq_{Int} f & \\
\Leftrightarrow & [\bar{\gamma} \circ [\bar{\alpha} \circ f]_{\sim_{AInt}}]_{\sim_{Int}} \sqsupseteq_{Int} f & [\ \text{by the defs. of } \gamma \text{ and } \alpha\ ] \\
\Leftrightarrow & \forall_h\ [h \cdot \bar{\gamma}\bar{\alpha}(f(h))] \supseteq [h \cdot f(h)] & [\ \text{by the def. of } \sqsubseteq_{Int}\ ] \\
\Leftrightarrow & \mathit{true} & [\ \bar{\gamma}\bar{\alpha}(\Theta) \supseteq \Theta\ ]
\end{array}
$$

5. $\alpha\gamma(g) = g$: follows directly from the definitions of α and γ and the fact that $\bar{\alpha}\bar{\gamma}(\kappa) = \kappa$.

□

4.3 Abstract semantics

An abstract semantics is defined in terms of an *abstract unification function* which is required to be *safe*.

DEFINITION 4.6. *safe abstract unification function*
Let P be a program, AInt a domain of abstract interpretations induced from ASub and $g \in AInt$. *We say that* $mgu_g^A : Atom^* \times Atom^* \to ASub$ *is* a safe abstract unification function *if for all* $\bar{a} \in Atom^*$ *and* $h \leftarrow \bar{b} \in P$

$$[h \cdot mgu_{\gamma(g)}(\bar{b}, \bar{a})] \subseteq [h \cdot \bar{\gamma}(mgu_g^A(\bar{b}, \bar{a}))].$$

Note that since the syntactic structure of elements of *ASub* is unspecified, we cannot rename apart abstract objects involving them. Instead, renaming apart is assumed to be handled in the definition of abstract unification. The following example illustrates a safe abstract unification function for *Dep* which is similar to that introduced and proved safe in [9]. Note that for any $\hat{g} \in (Atom \to Dep)/_\sim$ there is a representative $g \in Atom \to Dep$ such that for every atom h, $vars(g(h)) \subseteq vars(h)$. Hence it is straightforward to extend Υ to rename apart abstract objects, as assumed in the following.

EXAMPLE **6**. *Let AInt be induced from Dep and $g \in AInt$. Define $mgu_g^A : Atom^* \times Atom^* \to Dep$ as follows. Let $\langle a_1'\kappa_1, \ldots, a_n'\kappa_n\rangle = \Upsilon\langle a_1 g(a_1), \ldots, a_n g(a_n)\rangle$ in*

$$mgu_g^A(\langle b_1, \ldots, b_n\rangle, \langle a_1, \ldots, a_n\rangle) =$$
$$\bigcup_{i=1}^{n} \kappa_i \;\cup\; \left\{ (\{x\}, vars(t)) \,\middle|\, x \mapsto t \in mgu(\langle b_1, \ldots, b_n\rangle, \langle a_1', \ldots, a_n'\rangle) \right\}$$

For instance, if g maps the atom `append([x|x1],y,[x|z])` *to $\kappa = \{x1, y\} \leftrightarrow z$ and $\Upsilon\langle append([x \mid x1], y, [x \mid z])\kappa\rangle = \langle append([\bar{x} \mid \bar{x}1], \bar{y}, [\bar{x} \mid \bar{z}])[\{\bar{x}1, \bar{y}\} \leftrightarrow \bar{z}]\rangle$. Then*

$$mgu_g^A\left(\langle\, append(x, y, z)\,\rangle, \langle\, append([x \mid x1], y, [x \mid z])\,\rangle\right) = \begin{array}{l}[x \leftrightarrow \{\bar{x}, \bar{x}1\},\\ \quad y \leftrightarrow \bar{y},\\ \quad z \leftrightarrow \{\bar{x}, \bar{z}\},\\ \quad \{\bar{x}1, \bar{y}\} \leftrightarrow \bar{z}].\end{array}$$

While we do not introduce an explicit operator to restrict the result of the abstract unification to the variables in `append(x,y,z)`*, this functionality is captured by the equivalence induced by γ on AInt. Denoting $[x \leftrightarrow \{\bar{x}, \bar{x}1\}, y \leftrightarrow \bar{y}, z \leftrightarrow \{\bar{x}, \bar{z}\}, \{\bar{x}1, \bar{y}\} \leftrightarrow \bar{z}]$ by κ_1 we might say that the restriction of κ_1 to $\{x, y, z\}$ is $\kappa_2 = [\{x, y\} \leftrightarrow z]$ because $append(x, y, z) \cdot \bar{\gamma}(\kappa_1) \sim append(x, y, z) \cdot \bar{\gamma}(\kappa_2)$.*

In the sequel we assume that mgu_g^A denotes a safe abstract unification function.

DEFINITION **4.7**. *abstract semantic function*

$$F_P^A : AInt \to AInt$$

$$F_P^A(g) = \lambda_h \,.\, \bigsqcup \left\{ \kappa \,\middle|\, \begin{array}{l} h \leftarrow \bar{b} \in P,\ \bar{a} \in Atom^*, \\ \kappa = mgu_g^A(\bar{b}, \bar{a}) \end{array} \right\}$$

PROPOSITION **4.8**. *F_P^A is continuous and monotonic.*

Denote $[\![P]\!]_{abs} = lfp(F_P^A)$. The following theorem, which states the central result of this section, shows that $[\![P]\!]_{abs}$ indeed approximates the concrete semantics.

THEOREM **4.9**. *safety*
For any logic program P, $[\![P]\!]_{con} \sqsubseteq \gamma([\![P]\!]_{abs})$.

PROOF. The theorem is implied by (see e.g. [30])

$$\forall_{g \in AInt}\ F_P(\gamma(g)) \sqsubseteq \gamma(F_P^A(g))$$

which is equivalent, applying the definitions of F_P, F_P^A, $\sqsubseteq_{Int}$ and γ, to showing that for all $g \in AInt$ and $h \in Atom$:

$$\left[h \cdot \bigcup \left\{ \Theta \,\middle|\, \begin{array}{l} h \leftarrow \bar{b} \in P, \\ \bar{a} \in Atom^*, \\ \Theta = mgu_{\gamma(g)}(\bar{b}, \bar{a}) \end{array} \right\}\right] \subseteq \left[h \cdot \bar{\gamma}\left(\bigsqcup \left\{ \kappa \,\middle|\, \begin{array}{l} h \leftarrow \bar{b} \in P, \\ \bar{a} \in Atom^*, \\ \kappa = mgu_g^A(\bar{b}, \bar{a}) \end{array} \right\}\right)\right]$$

This follows easily from the safety of mgu_g^A (Definition 4.6) and monotonicity of $\bar{\gamma}$. □

4.4 Termination

In order to guarantee termination of analyses based on the abstract semantics described above, we must impose sufficient conditions to guarantee that F_P^A has finitely computable fixpoints. Standard conditions such as the requirement that *AInt* is *ascending chain finite*[3] (see e.g. [30]), are too restrictive. This is because our abstract domains are defined in terms of mappings from *Atom* to *ASub* and *Atom* is an infinite set. However, for a given program P, the functions $(F_P^A)^n(\bot)$ $(n \geq 0)$ only assign $\kappa \neq \bot_{ASub}$ to the atoms in $heads(P)$. Hence a sufficient condition on *ASub* is to require that for every $a \in Atom$, there are no infinite chains in *ASub* under the ordering induced by *AInt*:

PROPOSITION **4.10.** *termination*
Let AInt be a domain of abstract interpretations induced from ASub. Define for $a \in Atom$ the partial order $\sqsubseteq_a$ on ASub by $\kappa \sqsubseteq_a \kappa' \Leftrightarrow [a{\cdot}\bar{\gamma}(\kappa)] \subseteq [a{\cdot}\bar{\gamma}(\kappa')]$. If there are no infinite ascending $\sqsubseteq_a$ chains in ASub for any atom a then there exists a finite n such that $lfp(F_P^A) = (F_P^A)^n(\bot)$.

PROOF. Let $g_n = (F_P^A)^n(\bot)$ for $n \geq 0$ and for an atom h let $g_n^h : Atom \rightarrow ASub$ denote the abstract interpretation which maps h to $g_n(h)$ and all other atoms to $\bot_{ASub}$. The condition implies that for each $h \in heads(P)$, the chain $\{[g_n^h]_{\sim_{AInt}} \mid n \geq 0\}$ will be finite. By construction, $g_n = \bigsqcup_{h \in heads(P)} [g_n^h]_{\sim_{AInt}}$. Because $heads(P)$ is a finite set, the chain $g_0, g_1, g_2, \ldots$ will therefore also be finite. □

EXAMPLE **7.** *The domain $(2^{Sub}, \bar{\alpha}, Dep, \bar{\gamma})$ satisfies the condition of Proposition 4.10. To see this, observe that for any atom a and $\kappa \in Dep$ there exists $\kappa' \in Dep$ such that $vars(\kappa') \subseteq vars(a)$ and $a{\cdot}\gamma(\kappa) = a{\cdot}\gamma(\kappa')$. Hence if AInt is the domain of abstract interpretations induced from Dep and P is a logic program, then $F_P^A : AInt \rightarrow AInt$ has a finite least fixedpoint.*

5 Practical bottom-up analysis

The complexity of an analysis based on the framework described in the previous section is determined by the number of iterations it may take to reach a least fixed point of F_P^A and by the cost of each single iteration. The number of iterations is bounded by the height (i.e., the length of the longest chain) of the abstract domain *AInt*; the cost of one iteration depends on the number of abstract unifications it involves and on the complexity of each such unification (note that this operation is generic and hence we cannot assume anything about its complexity). For a program of size N, each iteration of F_P^A may involve $O(2^{(N+1)log\,N})$ abstract unifications as explained below. Although there exist more efficient ways to compute the least fixed point of a function (e.g., as in [28] where recomputing the results of previous iterations is avoided), the complexity of an analysis based on the evaluation of $lfp(F_P^A)$ is inherently exponential and hence not practical. In this section we present an alternative, the *join* semantic operator $F_P^{\mathcal{J}} : AInt \rightarrow AInt$, which involves $O(N^3)$ abstract unifications (albeit at the cost of accuracy).

[3] A complete lattice X is ascending chain finite if every nonempty subset $Y \subseteq X$ contains a maximal element; in this case every monotonic function defined on X has a finite ascending Kleene sequence.

5.1 Complexity

Let P be a program of size N, $g = (F_P^A)^k(\bot)$ and consider the number of abstract unifications needed to evaluate $(F_P^A(g))(h)$ for a clause $h \leftarrow b_1, \ldots, b_n$ in P (assuming for the present that h determines a unique clause). Figure 1 demonstrates the evaluation as specified in Definition 4.7. The top row contains the body of the clause, which should be (abstractly)

$$
\begin{array}{lllll}
\langle b_1 & \ldots & b_n \rangle & & \\
\langle a_1^1 & \ldots & a_n^1 \rangle & \rightarrow & \kappa^1 \\
\vdots & & & & \vdots \\
\vdots & & & & \vdots \\
\langle a_1^q & \ldots & a_n^q \rangle & \rightarrow & \kappa^q \\
 & & & & \sqcup \\
 & & & & \kappa
\end{array}
$$

Figure 1: Evaluation of $F_P^A(g)(h)$.

unified with each of the other rows. These consist of sequences of clause heads from the program. The κ^j to the right contain the respective results of the abstract unifications. Apart from the top row, there is a row for every combination of n clause heads from the program, so q is $O(N^N)$ as both n and the number of clauses in P are $O(N)$. The κ in the bottom right corner is defined by $\kappa = \sqcup\{\kappa^1, \ldots, \kappa^q\}$ and is the required value for $(F_P^A(g))(h)$. This demonstrates that evaluation of $(F_P^A(g))(h)$ might involve $O(N^N)$ (abstract) unifications and hence, each iteration $O(N^{(N+1)})$ or $O(2^{(N+1)\log N})$ unifications.

The above analysis is pessimistic. Usually the maximal number of atoms occurring in the body of a clause is bounded by a constant K. In that case the complexity is reduced to $O(N^{(K+1)})$ which is polynomial. Furthermore, if the number of clauses defining a predicate is also bounded, then we get a linear complexity. However, since these constants might be large, we prefer an algorithm that **guarantees** better complexity.

5.2 A join approximation

The idea behind the definition of the alternative $F_P^{\mathcal{J}} : AInt \rightarrow AInt$ is motivated by the observation that the $O(N^N)$ rows of Figure 1 contain only $O(N)$ distinct atoms. We would like to exploit this fact by decomposing the abstract unification of a row $a_1^j, \ldots, a_n^j$ with $b_1, \ldots, b_n$ into n unifications of a_i^j with b_i. But instead of composing the results for each row to evaluate the corresponding κ^j (which would again involve $O(N^N)$ compositions) we first take the join for each column and then compose the resulting abstract substitutions. This is illustrated in Figure 2 where we assume that $heads(P) = \{h_1, \ldots, h_r\}$ and denote $\bar{\kappa}_i^j = mgu_g^A(\langle b_i \rangle, \langle h_j \rangle)$ $(1 \leq i \leq n, \quad 1 \leq j \leq r)$. The $\hat{\kappa}_i$ $(1 \leq i \leq n)$ are defined by $\hat{\kappa}_i = \sqcup\{\bar{\kappa}_i^j \mid 1 \leq j \leq r\}$. Note how the number of rows changed from q in Figure 1 to r here. The result κ' is evaluated by *composing* the $\hat{\kappa}_i$ $(1 \leq i \leq n)$. This composition is captured

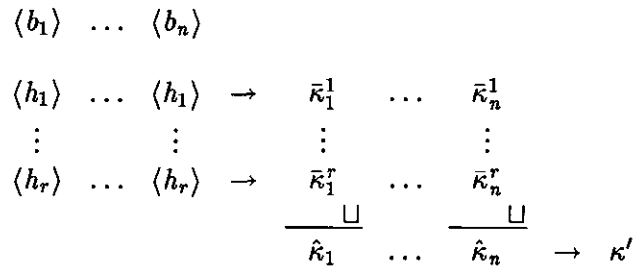

Figure 2: Evaluation of $F_P^{\mathcal{J}}(g)(h)$.

by defining: $\kappa' = mgu_{\hat{g}}^{A}(\langle b_1, \ldots, b_n\rangle, \langle b_1, \ldots, b_n\rangle)$ where $\hat{g}$ maps b_i to $\hat{\kappa}_i$ for $1 \leq i \leq n$ (see comment after Definition 3.4).

Thus, the evaluation of κ' involves $(n \cdot r) + 1$ abstract unifications. This amounts to $O(N^2)$ unifications for each clause and $O(N^3)$ unifications for one iteration of $F_P^{\mathcal{J}}(g)$.

After introducing a formal definition we justify the safety of this approach in Theorem 5.3 below.

DEFINITION **5.1**. *joined semantic function*
Let P be a logic program. Define $F_P^{\mathcal{J}} : AInt \to AInt$ by:

$$F_P^{\mathcal{J}}(g) = \lambda_h \cdot \bigsqcup \left\{ \kappa \;\middle|\; \begin{array}{l} h \leftarrow b_1, \ldots, b_n \in P, \;\; 1 \leq i \leq n, \\ \hat{\kappa}_i = \sqcup\{mgu_g^{A}(b_i, a) \mid a \in heads(P)\}, \\ \kappa = mgu_{\hat{g}}^{A}(\langle b_1, \ldots, b_n\rangle, \langle b_1, \ldots, b_n\rangle) \end{array} \right\}$$

where $\hat{g}$ maps b_i to $\hat{\kappa}_i$ for $1 \leq i \leq n$.

PROPOSITION **5.2**.
$F_P^{\mathcal{J}}$ is continuous and monotonic.

Denote $[\![P]\!]_{join} = lfp(F_P^{\mathcal{J}})$. The following theorem shows that $[\![P]\!]_{join}$ indeed approximates the concrete semantics.

THEOREM **5.3**. *safety*
For any logic program P, $[\![P]\!]_{con} \sqsubseteq \gamma([\![P]\!]_{join})$.

PROOF. See appendix. □

EXAMPLE **8**. *Let P be the program from Figure 3 which specifies the quicksort relation. Let $AInt$ be the domain of abstract interpretations induced from Dep and let $g \in AInt$ be the abstract interpretation described by Table 1 below. In fact, $g = [\![P]\!]_{join}$ and hence the*

```
qs([ ],[ ]).
qs([x|u],y) :-
     split(x,u,v,w),
     qs(v,v1),
     qs(w,w1),
     append(v1,[x|w1],y).

gt(s(0),0).
gt(s(x),s(y)) :- gt(x,y).

le(0,0).
le(0,s(0)).
le(s(x),s(y)) :- le(x,y).

split(x,[ ],[ ],[ ]).
split(x,[u|u1],[u|v],w) :-
     gt(x,u),
     split(x,u1,v,w).
split(x,[u,|u1],v,[u|w]) :-
     le(x,u),
     split(x,u1,v,w).

append([ ],y,y).
append([x|x1],y,[x|z]) :-
     append(x1,y,z).
```

Figure 3: A logic program for quicksort.

table summarizes the ground dependency analysis for P. Consider the second clause in the definition of the predicate $qs/2$:

$$\texttt{qs}([\texttt{x} \mid \texttt{u}], \texttt{y}) : -\texttt{split}(\texttt{x}, \texttt{u}, \texttt{v}, \texttt{w}), \texttt{qs}(\texttt{v}, \texttt{v1}), \texttt{qs}(\texttt{w}, \texttt{w1}), \texttt{append}(\texttt{v1}, [\texttt{x} \mid \texttt{w1}], \texttt{y}).$$

Evaluating $F_P^A(g)(qs([x \mid u], y))$ involves 24 abstract unifications as there are $3 \times 2 \times 2 \times 2$ sequences of heads in P which match the clause body. Evaluation of $F_P^J(g)(qs([x \mid u], y))$ involves 10 abstract unifications and is carried out as follows[4]*:*

1. *abstract unification of* `split(x,u,v,w)` *with the table entries 3,4,5 gives*[5]*:*

$$\hat{\kappa}_1 = [u \leftrightarrow v \leftrightarrow w \leftrightarrow \emptyset] \;\sqcup_{Dep}\; [x \leftrightarrow u \leftrightarrow v \leftrightarrow w \leftrightarrow \emptyset] \;=\; [u \leftrightarrow v \leftrightarrow w \leftrightarrow \emptyset];$$

2. *abstract unification of* `qs(v,v1)` *with the table entries 1,2 gives:*

$$\hat{\kappa}_2 = [v \leftrightarrow v1 \leftrightarrow \emptyset] \;\sqcup_{Dep}\; [v \leftrightarrow v1] = [v \leftrightarrow v1];$$

3. *abstract unification of* `qs(w,w1)` *with the table entries 1,2 gives:*

$$\hat{\kappa}_3 = [w \leftrightarrow w1 \leftrightarrow \emptyset] \;\sqcup_{Dep}\; [w \leftrightarrow w1] = [w \leftrightarrow w1];$$

4. *abstract unification of* `append(v1,[x|w1],y)` *with the table entries 6,7 gives:*

$$\hat{\kappa}_4 = [v1 \leftrightarrow \emptyset, \{x, w1\} \leftrightarrow y] \;\sqcup_{Dep}\; [\{v1, x, w1\} \leftrightarrow y] = [\{v1, x, w1\} \leftrightarrow y];$$

[4] To simplify the presentation, abstract substitutions are restricted to the relevant variables.

[5] Note that the join on *Dep* is defined by $R \sqcup_{Dep} R' = dep(R) \cap dep(R')$.

clause no.	clause head	abstract substitution
1.	`qs([ ],[ ])`	$\emptyset$
2.	`qs([x|u],y)`	$u \leftrightarrow \emptyset, x \leftrightarrow y$
3.	`split(x,[ ],[ ],[ ])`	$\emptyset$
4.	`split(x,[u|u1],[u|v],w)`	$x \leftrightarrow u \leftrightarrow u1 \leftrightarrow v \leftrightarrow w \leftrightarrow \emptyset$
5.	`split(x,[u,|u1],v,[u|w])`	$x \leftrightarrow u \leftrightarrow u1 \leftrightarrow v \leftrightarrow w \leftrightarrow \emptyset$
6.	`append([ ],y,y)`	$\emptyset$
7.	`append([x|x1],y,[x|z])`	$\{x1, y\} \leftrightarrow z$
8.	`gt(s(0),0)`	$\emptyset$
9.	`gt(s(x),s(y))`	$x \leftrightarrow y \leftrightarrow \emptyset$
10.	`le(0,0)`	$\emptyset$
11.	`le(0,s(0))`	$\emptyset$
12.	`le(s(x),s(y))`	$x \leftrightarrow y \leftrightarrow \emptyset$

Table 1: Ground dependencies for quicksort.

5. $mgu_{\hat{g}}^{\mathcal{A}}\left(\left\langle \begin{array}{l} split(x,u,v,w), \\ qs(v,v1), \\ qs(w,w1), \\ append(v1,[x \mid w1],y) \end{array} \right\rangle, \left\langle \begin{array}{l} split(x,u,v,w), \\ qs(v,v1), \\ qs(w,w1), \\ append(v1,[x \mid w1],y) \end{array} \right\rangle\right) = [u \leftrightarrow \emptyset, x \leftrightarrow y]$

where $\hat{g}$ *maps* `split(x,u,v,w)` *to* $\hat{\kappa}_1$, `qs(v,v1)` *to* $\hat{\kappa}_2$, `qs(w,w1)` *to* $\hat{\kappa}_3$ *and* `append(v1,[x|w1],y)` *to* $\hat{\kappa}_4$.

5.3 Approximating answer substitutions

In order to provide approximations of the answer substitutions for a logic program with an initial goal, we provide the following:

DEFINITION 5.4. *abstract answers*
Let AInt be a domain of abstract interpretations induced from ASub. The abstract answers for a given goal and a program P which are determined by $g \in AInt$ *are specified by the function* $ans_g^{\mathcal{A}} : Atom^* \rightarrow ASub$ *defined by*

$$ans_g^{\mathcal{A}}\langle b_1, \ldots, b_n\rangle = mgu_{\hat{g}}^{\mathcal{A}}(\langle b_1, \ldots, b_n\rangle, \langle b_1, \ldots, b_n\rangle)$$

where $\hat{g}$ *maps* b_i *to* $\sqcup\{mgu_g^{\mathcal{A}}(b_i, h) \mid h \in heads(P)\}$ *for* $1 \leq i \leq n$.

The following theorem provides the basis for approximating the answer substitutions for a program P with an initial goal G.

THEOREM **5.5.** *Let AInt be a domain of abstract substitutions induced from ASub. Let P be a logic program. Then for any goal G,*

$$[G \cdot ans_{[\![P]\!]_{con}}(G)] \subseteq [G \cdot \bar{\gamma}\left(ans^{A}_{[\![P]\!]_{join}}(G)\right)].$$

PROOF. The proof of the more general theorem which states that for any $g \in AInt$ and $f \in Int$ such that $f \sqsubseteq_{Int} \gamma(g)$, $[G \cdot ans_f(G)] \subseteq [G \cdot \bar{\gamma}(ans^{A}_{g}(G))]$, is similar to that of Theorem 5.3. □

EXAMPLE **9.** *Consider the quicksort program from Example 8 with the goal* $qs([3,1,2],x)$.

$$ans^{A}_{[\![P]\!]_{join}}(qs([3,1,2],x)) = [x \leftrightarrow \emptyset]$$

which specifies that any answer for this goal binds x *to a ground term.*

6 Modeling Control

Standard semantics typically associate programs with entities which capture the essence of their behaviour while abstracting away details related to the text of the program as well as the control of the execution model. For semantics-based program analysis an enhanced or *"collecting"* semantics which recaptures some of these details is usually required. After all, the purpose of program analysis is often to analyse the *text* of the program with respect to the *control* of its execution model. In general, the fact that collecting semantics can be viewed as uncovering details which the standard semantics has hidden imposes a restriction on the choice of semantic models upon which program analyses can be based.

In the case of logic programs, standard semantics traditionally associate programs with the set of ground atoms which they imply. Program analyses, in contrast, are often required to capture: (1) answer substitutions for a query; and (2) call patterns, which provide information about how particular clauses in the program are used in refutations of a query. It is no coincidence that in most cases practical abstract interpretations of logic programs approximate top-down semantics based on SLD resolution (e.g., [6]). The information concerning control and textual details of a program are more naturally recovered (and collected) from such semantics.

In this paper we have first introduced a bottom-up semantics (basically that of [13]) which captures answer substitutions and we demonstrated how abstract interpretations can provide approximations of answer substitutions. In this section we are concerned with analyses which capture more (control) details of a computation. In particular we show how to approximate (a) the partial answers, and (b) the call patterns of a goal. However, instead of enhancing the semantics, we propose to enhance the program so that its standard meaning reflects the additional information required. The key idea is to enhance a program P by a transformation M so that the standard meaning of $M(P)$ reflects the additional information to be collected by an analysis of P. This idea is further investigated in [2].

6.1 Abstracting for partial answer substitutions

Concurrent logic programs are characterized not by their successful computations alone but also by computations which *fail, suspend* or *diverge.* To fully characterize such programs, additional sequencing and branching information is required. A rough approximation can be obtained by ignoring synchronization, viewing a concurrent logic program as a pure logic program. However, in this case analyses should consider all computations (i.e., success, fail, suspend, diverge).

In this section we demonstrate how to provide approximations of the *partial answer substitutions* of a logic program. Partial answer substitutions are substitutions which correspond to the partial computations of the goal, regardless if they eventually succeed, fail, suspend or diverge. Ignoring synchronization implies that analyses based on our framework cannot reason about reactive properties of concurrent programs. However, it is useful for a wide range of applications that do not focus on such properties.

DEFINITION **6.1**. *partial answer substitutions*
Let P be a program. We say that $\theta \in Sub$ is a partial answer substitution (or partial answer for short) for a goal G if $\langle G; \epsilon \rangle \rightarrow^* \langle G'; \theta' \rangle$ and $\theta = \theta' \restriction vars(G)$.

Falaschi and Levi [12] show that the partial answer substitutions of a logic program P can be determined by adding to P an additional unit clause for each predicate in P and considering the answer substitutions of the transformed program.

DEFINITION **6.2**. *transformed program $P_G^{\perp}$*
Let P be a program and G a goal. We denote by $P_G^{\perp}$ the program constructed by adding to P a clause of the form $p(x_1, \ldots, x_n)$ for every predicate p/n in P_G where $x_1, \ldots, x_n$ are distinct variables.

PROPOSITION **6.3**.
Let P be a logic program. Then $\theta \in Sub$ is a partial answer for a goal G iff there exists $\theta' \in ans_{[P_G^{\perp}]_{con}}(G)$ such that $G\theta \sim G\theta'$.

PROOF. See [12]. □

EXAMPLE **10**.
Consider the following program P with the goal $G = p(x)$:

```
p([a|x]) :- p(x).
```

The corresponding transformed program $P_G^{\perp}$ is:

```
p([a|x]) :- p(x).
p(_).
```

The goal G has no answer substitutions in P. However, its partial answers:

$$\{\epsilon, \{x \mapsto [a \mid \_]\}, \{x \mapsto [a, a \mid \_]\}, \{x \mapsto [a, a, a \mid \_]\}, \ldots\}$$

can be obtained as $ans_{[P_G^{\perp}]_{con}}(G)$.

The partial answer substitutions for a program with initial goal P_G can be approximated by approximating the answer substitutions of $P_G^{\perp}$. In particular, $ans^{\mathcal{A}}_{[\![P_G^{\perp}]\!]_{join}}$ will provide such an approximation.

COROLLARY **6.4.** *safety*
Let AInt be a domain of abstract interpretations induced from ASub and let P be a logic program. If $\theta \in Sub$ is a partial answer for a goal G then there exists $\theta' \in \bar{\gamma}(ans^{\mathcal{A}}_{[\![P_G^{\perp}]\!]_{join}}(G))$ such that $G\theta \sim G\theta'$.

PROOF. Immediate from Proposition 6.3 and Theorem 5.5. □

EXAMPLE **11.** *Let $(Int, \alpha, AInt, \gamma)$ be the domain of abstract interpretations induced from $(2^{Sub}, \bar{\alpha}, Sym, \bar{\gamma})$ (defined in Example 4). For $g \in AInt$ let $mgu_g^{\mathcal{A}} : Atom^* \times Atom^* \to Sym$ be defined by $mgu_g^{\mathcal{A}}(\bar{b}, \bar{a}) = \bar{\alpha}(mgu_{\gamma(g)}(\bar{b}, \bar{a}))$. Consider the program P from the previous example:*

```
p([a|x]) :- p(x).
```

The abstract meaning $[\![P^{\perp}]\!]_{join}$ maps `p([a|x])` *to {'a','|'}. The partial answers of the goal $p(x)$ are approximated by $\kappa = \{'a','|'\}$ indicating that any partial answer for $p(x)$ binds x to a term containing only those function symbols. Hence, since the arity of '|' is 2 and the arity of 'a' is 0, x is bound to a tree in which all leaves are either 'a' or variables.*

6.2 Abstracting for call patterns

Often, analyses of logic programs are required to provide in addition to the success patterns of a program (and goal) also its *call patterns*. Call patterns determine how the clauses of a program will be "used" in computations; such information may for example provide the basis for program specialization and optimization.

Top-down semantics for logic programs are readily extended to collect call patterns as the evaluation of the recursive semantic functions usually corresponds to the operational behaviour of programs. Typically, an additional argument can simply be added to the semantic equations and used to accumulate the sets of call patterns which correspond to the calls which arise in actual computations. OLDT resolution ([32]) is an example of such a semantics which extends SLD resolution by recording the calls arising in computations of the initial goal together with their answer substitutions (if any).

In bottom-up semantics there is no corresponding notion of calls, and extension of these semantics to collecting interpretations is not as straightforward. However, there exist methods to transform a logic program so that the bottom-up evaluation of the transformed program corresponds to the operational (top-down) behaviour of the original program. The Magic Set and Alexander methods ([5, 1, 27]) are such techniques, which appeared first in the context of deductive databases. Bry shows in [7] that both collecting semantics like OLDT and these transformation methods are instances of the same fixpoint semantics, which is called the Backward Fixpoint Procedure and defined in terms of a meta-interpreter.

Here we apply the Magic Set method to capture both calls and answers for a program with an initial goal. We consider two different computational models, distinguished by their computation rule. Recall that the answers for a given goal are independent of the computational rule. However, this is not the case for the set of calls which arise in computations. The semantic model which is based on a non-deterministic computation rule (which is the one that we have been considering so far, see Section 2) is useful for approximating the behaviour of concurrent logic programs. The semantics with a left-to-right computation rule is defined similarly, by choosing the leftmost atom from the goal for every reduction. This model approximates better the execution model of Prolog. We sometimes distinguish between the two models by referring to programs as "sequential logic programs" or "pure logic programs", depending on whether we assume, respectively, a left-to-right or a non-deterministic computation rule.

DEFINITION **6.5.** *call patterns*
Let P be a (sequential or pure) logic program and G a goal. We say that an instance $a\varphi$ of an atom $a \in atoms(P_G)$ is a call pattern (or a call for short) if

$$\langle G; \epsilon \rangle \rightarrow^* \langle \ldots, a, \ldots; \varphi \rangle \stackrel{a\varphi}{\rightarrow}$$

where the label $a\varphi$ on the transition arrow denotes the atom selected by the computation rule.

The following definition describes how the Magic Set method transforms a program with initial goal P_G into the magic program $P_G^{\mathcal{M}}$.

DEFINITION **6.6.** *magic program $P_G^{\mathcal{M}}$*
Let P_G be a program with initial goal $G = a_1, \ldots, a_n$. The magic program $P_G^{\mathcal{M}}$ is obtained by transforming P_G as follows.

- *For sequential logic programs:*

  (s1) replace G by the clauses $a_i^{\mathcal{C}} \leftarrow a_1^{\mathcal{A}}, \ldots, a_{i-1}^{\mathcal{A}}$ for $1 \leq i \leq n$;

  (s2) replace each clause $h \leftarrow b_1, \ldots, b_m \in P$ $(m \geq 0)$ by the clauses $b_i^{\mathcal{C}} \leftarrow h^{\mathcal{C}}, b_1^{\mathcal{A}}, \ldots, b_{i-1}^{\mathcal{A}}$ for $1 \leq i \leq m$ and $h^{\mathcal{A}} \leftarrow h^{\mathcal{C}}, b_1^{\mathcal{A}}, \ldots, b_m^{\mathcal{A}}$.

- *For pure logic programs:*

  (p1) replace G by the clauses $a_i^{\mathcal{C}} \leftarrow a_1^{\mathcal{P}}, \ldots, a_{i-1}^{\mathcal{P}}, a_{i+1}^{\mathcal{P}}, \ldots, a_n^{\mathcal{P}}$ for $1 \leq i \leq n$;

  (p2) replace each clause $h \leftarrow b_1, \ldots, b_m \in P$ $(m \geq 0)$ by the clauses $b_i^{\mathcal{C}} \leftarrow h^{\mathcal{C}}, b_1^{\mathcal{P}}, \ldots, b_{i-1}^{\mathcal{P}}, b_{i+1}^{\mathcal{P}}, \ldots, b_m^{\mathcal{P}}$ for $1 \leq i \leq m$ and $h^{\mathcal{P}} \leftarrow h^{\mathcal{C}}, b_1^{\mathcal{P}}, \ldots, b_m^{\mathcal{P}}$;

  (p3) for each predicate $p^{\mathcal{P}}/n$ in the program obtained by applying the rules (p1) and (p2), add the fact $p^{\mathcal{P}}(x_1, \ldots, x_n)$, where $x_1, \ldots, x_n$ are distinct variables.

The $\mathcal{C}$-, $\mathcal{A}$- and $\mathcal{P}$-annotations on atoms are just labels, so, e.g., $vars(p) = vars(p^{\mathcal{C}})$ for atom p. An annotated atom $p^{\mathcal{C}}$ is read as "p is a call"; the atoms $p^{\mathcal{A}}$ and $p^{\mathcal{P}}$ are interpreted as "p has an answer substitution" and "p has a partial answer substitution" respectively. So, for

example, the first transformed clause in rule (s2) in the above definition can be informally read as: "b_i *is a call if* h *is a call and there are answers for* b_1 *till* b_{i-1}". Note that the Magic Set transformation leads to a quadratic increase in size.

EXAMPLE **12.**
Let P_G *be the initialized sequential logic program*

```
← p(a).

p(x) ← q(x), p(f(x)).
```

Then the transformed program $P_G^{\mathcal{M}}$ *is*

```
p^C(a).


q^C(x) ← p^C(x).

p^C(f(x)) ← p^C(x), q^A(x).

p^A(x) ← p^C(x), q^A(x), p^A(f(x)).
```

The first clause in $P_G^{\mathcal{M}}$ *reads as "*$p(a)$ *is a call"; the third clause is read "*$p(f(x))$ *is a call if* $p(x)$ *is a call and* $q(x)$ *has an answer substitution".*

The concrete meaning of $P_G^{\mathcal{M}}$ *provides* $atoms([\![P_G^{\mathcal{M}}]\!]_{con}) = \{p^{\mathcal{C}}(a), q^{\mathcal{C}}(a)\}$, *while the calls of* P_G *are* $\{p(a), q(a)\}$. *Note that* $p(f(a))$ *is not a call because* $q(a)$ *does not have an answer substitution. The fact that* P_G *has no answer substitutions (note that it has no facts) is reflected by the fact that* $atoms([\![P_G^{\mathcal{M}}]\!]_{con})$ *does not contain atoms of the form* $p^{\mathcal{A}}$ *or* $q^{\mathcal{A}}$.

It has been proven in [7, 5, 29] that the Magic Set and Alexander methods are sound and complete proof procedures for ground instances of calls and answers of sequential logic programs: for every atom a in the Herbrand interpretation of the original program, there is a corresponding atom $a^{\mathcal{A}}$ for the magic program, and reversely. We extend this result for non-ground atoms[6] and pure logic programs; furthermore, we need the property that the bottom-up evaluation of the magic program indeed corresponds to top-down execution of the original program, in the sense that for every call a in the top-down evaluation of the original program, there is a corresponding atom $a^{\mathcal{C}}$ represented by the meaning of the transformed program. These extensions are reflected in the theorem below, the proof of which can be found in the appendix.

THEOREM **6.7.** *soundness of magic*
Consider a partial computation of a (sequential or pure) logic program P *with initial goal* G. *Let* $a \in atoms(P_G)$ *and* $\varphi \in Sub$ *and suppose that* $a\varphi$ *is a call in this computation. Then:*

1. $a^{\mathcal{C}}\varphi \in atoms([\![P_G^{\mathcal{M}}]\!]_{con})$;
2. (a) *if* P *is a sequential logic program and* σ *is an answer substitution for* $a\varphi$, *then* $\sigma \in ans_{[\![P_G^{\mathcal{M}}]\!]_{con}}(a^{\mathcal{A}}\varphi)$;

[6] Bry [7] claims that pre-encoding of variables can be applied to extend these results for non-ground instances. However, as demonstrated in Example 13 (below), completeness does not always hold. Also Ramakrishnan ([25]) considers a non-ground case, however, the proof is lacking.

(b) if P is a pure logic program and σ is a partial answer substitution for $a\varphi$, then $\sigma \in ans_{[P_G^{\mathcal{M}}]_{con}}(a^{\mathcal{P}}\varphi)$.

The other direction of the theorem (completeness) does not hold. The following example shows that not every atom of the form $p^{\mathcal{C}}$ represented by the meaning of $P_G^{\mathcal{M}}$ necessarily corresponds to a call in P:

EXAMPLE **13.** *counter example*
Let P_G be the initialized sequential logic program:

```
← q.

q ← p(a), p(x), r(x).

p(x).
```

The transformed program $P_G^{\mathcal{M}}$ is

```
q^C.

p^C(a) ← q^C.

p^C(x) ← q^C, p^A(a).

r^C(x) ← q^C, p^A(a), p^A(x).

q^A ← q^C, p^A(a), p^A(x), r^A(x).

p^A(x) ← p^C(x).
```

The calls that arise in the computation of P_G are $\{q, p(a), p(x), r(x)\}$. However, the bottom-up meaning of $P_G^{\mathcal{M}}$ indicates also a call of the form $r(a)$.

This example demonstrates an essential difference between methods such as OLDT resolution and the Magic Set approach. On one hand, the magic approach does cause the evaluation of the bottom-up semantic function to correspond more closely to the operational behaviour of a program. However, while OLDT resolution specifies pairs of calls and corresponding answer substitutions, the bottom-up semantics of the magic program specifies a set of calls and a set of atoms which determine the answers for arbitrary goals. While OLDT resolution computes answers only for those goals which are called in the course of a computation (for an initial goal), this does not carry over precisely through the Magic Set transformation (although it is safe and in most cases sufficiently accurate).

A more precise approach involves modifying the Magic Set transformation replacing rules (s2) and (p2) in Definition 6.6 by:

(s2′) replace each clause $h \leftarrow b_1, \ldots, b_m \in P$ $(m \geq 0)$ by the clauses $b_i^{\mathcal{C}} \leftarrow h^{\mathcal{C}}, b_1^{\mathcal{A}}, \ldots, b_{i-1}^{\mathcal{A}}$ for $1 \leq i \leq m$ and $h^{\mathcal{A}} \leftarrow b_1^{\mathcal{A}}, \ldots, b_m^{\mathcal{A}}$;

(p2′) replace each clause $h \leftarrow b_1, \ldots, b_m \in P$ $(m \geq 0)$ by the clauses $b_i^{\mathcal{C}} \leftarrow h^{\mathcal{C}}, b_1^{\mathcal{P}}, \ldots, b_{i-1}^{\mathcal{P}}, b_{i+1}^{\mathcal{P}}, \ldots, b_m^{\mathcal{P}}$ for $1 \leq i \leq m$ and $h^{\mathcal{P}} \leftarrow b_1^{\mathcal{P}}, \ldots, b_m^{\mathcal{P}}$.

In this approach, evaluation of the bottom-up semantics no longer corresponds to the operational behaviour of the original program. Furthermore, the bottom-up evaluation of the (concrete or abstract) meaning of a transformed program is less efficient as the least fixed point evaluates all of the implied instances of the original program. However, the transformed programs now determine precisely the answer and partial answer substitutions for any goal. Note that the transformed clauses of the form $h^{\mathcal{A}} \leftarrow b_1^{\mathcal{A}}, \ldots, b_m^{\mathcal{A}}$ and $h^{\mathcal{P}} \leftarrow b_1^{\mathcal{P}}, \ldots, b_m^{\mathcal{P}}$ are isomorphic to the original program clauses. Proving soundness (for the set of calls determined) in this approach is a simplification of the proof of Theorem 6.7. We conjecture that completeness holds in this case. That is, the set of calls determined contains precisely those that arise in computations.

Given that the call patterns for a program P_G are captured by the answer substitutions of the magic program $P_G^{\mathcal{M}}$, we can approximate the set of calls by the framework described in the previous sections. In particular, if a is a specific occurrence of an atom in a program, we can approximate the ways that a will be activated as a call in the computations of P_G, as expressed by the following corollary.

COROLLARY **6.8.** *safety*
For a program P with initial goal G and $a \in atoms(P_G)$:

$$a\theta \text{ is a call in } P_G \quad \Rightarrow \quad a^{\mathcal{C}}\theta \in atoms(\gamma([\![P_G^{\mathcal{M}}]\!]_{join}))$$

PROOF.

$$\begin{array}{lll} & a\theta \text{ is a call in } P_G & \\ \Rightarrow & a^{\mathcal{C}}\theta \in atoms([\![P_G^{\mathcal{M}}]\!]_{con}) & [\ \text{by Theorem 6.7}\ ] \\ \Rightarrow & a^{\mathcal{C}}\theta \in atoms(\gamma([\![P_G^{\mathcal{M}}]\!]_{join})) & [\ \text{by Theorem 5.3}\ ] \end{array}$$

□

EXAMPLE **14.**
Consider the quicksort (sequential) program from Figure 3 with an initial goal of the form $qs(z1, z2)$ ($z1$ and $z2$ are bound to arbitrary terms):

```
← qs(z1,z2).

qs([],[]).

qs([x|u],y) ← split(x,u,v,w), qs(v,v1),
              qs(w,w1), append(v1,[x|w1],y).
```

The magic program includes the following clauses:

```
qs^C(z1,z2).

qs^C(v,v1) ← qs^C([x|u],y), split^A(x,u,v,w).

qs^C(w,w1) ← qs^C([x|u],y), split^A(x,u,v,w), qs^A(v,v1).
```

Applying the ground dependency abstraction defined in Example 3 to approximate the answer substitutions of the transformed program provides the information that in any activation of the second $qs/2$ clause, the call $qs(v, v1)$ has variable v ground. Furthermore, since the variable $v1$ does not occur in the body of the corresponding transformed clause we may infer that it is uninstantiated. A similar argument holds for the call $qs(w, w1)$ so that we may derive that the calls $qs(v, v1)$ and $qs(w, w1)$ are "independent" (and can be executed in parallel [22]).

7 Conclusions

We have presented a formal framework for the bottom-up abstraction of sequential and concurrent logic programs which is suitable for analysing answer substitutions, partial answer substitutions and call patterns. The framework is based on a bottom-up semantics for logic programs which evolves from that defined by Falaschi *et al.* [13]. This semantics was first applied in the context of abstract interpretation by Barbuti *et al.* [3, 4]. It was independently used by us in [8] and by Kemp and Ringwood in [16]. The semantics of [13] provides an attractive basis for abstraction due to its simplicity, its similarity to the standard T_P semantics and due to the correspondence to the operational semantics (namely answer substitutions) which is further discussed in [14]. Marriott and Søndergaard also introduce a bottom-up semantics in [19] where they sketch an example dataflow analysis based on its abstraction.

The main contribution of this paper is in the extensions of the bottom-up framework to approximate partial answer substitutions and call patterns. The first extension applies a result of Falaschi and Levi [12]. They show that a program can be augmented so that the answer substitutions of the augmented program correspond to the partial answer substitutions of the original program. We apply this result in defining a safe approximation to partial answers. Approximations of this type are useful as a basis for analyses of concurrent logic programs which are not concerned with reactive aspects.

A similar strategy is followed to evaluate call patterns in a bottom-up framework, by extending programs with Magic Sets [5]. This idea is already suggested by Marriott and Søndergaard in [19]. Our result basically shows that the bottom-up semantics of logic programs can be extended to a collecting semantics which approximates both success and call patterns. To the best of our knowledge, a proof of safety of the Magic Set transformation for abstract interpretation (see Theorem 6.7) has not been previously published. Since the submission of this paper, similar results have been reported by Nilsson [24], by Debray and Ramakrishnan [11] and by Ramakrishnan [26]. Furthermore, the conjecture made in Section 6.2 concerning completeness of the modified Magic Set transformation has recently been proven by Steiner [31].

Acknowledgements

We acknowledge Moreno Falaschi, Rob Gerth and Giorgio Levi for their helpful comments. Kim Marriott suggested to us the idea of applying the Magic Set method. Example 13 is due to John Gallagher who deserves special thanks for his thorough review of an earlier version of this paper. The comments and suggestions of the anonymous referees are appreciated.

Appendix

LEMMA 1. *Let $(2^{Sub}, \bar{\alpha}, ASub, \bar{\gamma})$ be a domain of abstract substitutions (see Definition 4.2). Then for any atom $a \in Atom$ and $\Theta_1, \Theta_2 \in 2^{Sub}$:*

$$[a \cdot \Theta_1] \subseteq [a \cdot \Theta_2] \;\Rightarrow\; [a \cdot \bar{\gamma}\bar{\alpha}(\Theta_1)] \subseteq [a \cdot \bar{\gamma}\bar{\alpha}(\Theta_2)]$$

PROOF. Assume the premise of the lemma and let $\Theta_2' \subseteq \Theta_2$ be such that $a \cdot \Theta_1 \sim a \cdot \Theta_2$ (such a Θ_2' always exists). Then we have by Definition 4.2 that $[a \cdot \bar{\gamma}\bar{\alpha}(\Theta_1)] = [a \cdot \bar{\gamma}\bar{\alpha}(\Theta_2')]$ and by monotonicity of $\bar{\alpha}$ and $\bar{\gamma}$ that $[a \cdot \bar{\gamma}\bar{\alpha}(\Theta_2')] \subseteq [a \cdot \bar{\gamma}\bar{\alpha}(\Theta_2)]$. □

THEOREM **5.3**
For any logic program P, $\quad [\![P]\!]_{con} \sqsubseteq \gamma([\![P]\!]_{join})$.

PROOF. As in the proof of Theorem 4.9, we show that

$$\forall_{g \in AInt}\; F_P(\gamma(g)) \sqsubseteq \gamma(F_P^{\mathcal{J}}(g)).$$

Throughout the proof, we denote $\bar{b} = b_1, \ldots, b_n$, $\bar{a} = a_1, \ldots, a_n$ and let the index i range between 1 and n. We show that for every $g \in AInt$ and $h \in Atom$:

$$\left[h \cdot \bigcup \left\{ \Theta \;\middle|\; \begin{array}{l} h \leftarrow \bar{b} \in P, \\ \bar{a} \in heads(P)^n, \\ \Theta = mgu_{\gamma(g)}(\bar{b}, \bar{a}) \end{array} \right\}\right] \subseteq \left[h \cdot \bar{\gamma}\left(\bigsqcup \left\{ \kappa \;\middle|\; \begin{array}{l} h \leftarrow \bar{b} \in P,\ 1 \le i \le n, \\ \hat{\kappa}_i = \sqcup\{mgu_g^A(b_i, a) \mid a \in heads(P)\}, \\ \kappa = mgu_{\hat{g}}^A(\bar{b}, \bar{b}) \end{array} \right\}\right)\right] \qquad (1)$$

where $\hat{g}$ maps b_i to $\hat{\kappa}_i$ for $1 \le i \le n$.

Take an element $[h\theta]$ in the left-hand side of equation (1). So there exist $h \leftarrow \bar{b} \in P$ and $\bar{a} \in heads(P)^n$ such that $\theta \in mgu_{\gamma(g)}(\bar{b}, \bar{a})$ which implies that $\theta = mgu(\langle b_1, \ldots, b_n\rangle, \Upsilon\langle a_1\theta_1, \ldots, a_n\theta_n\rangle)$ where $\theta_i \in \gamma(g)(a_i)$. The proof proceeds in two steps:

1. Let $\psi_i = mgu(\langle b_i\rangle, \Upsilon\langle a_i\theta_i\rangle)$ and $\sigma = mgu(\langle b_1, \ldots, b_n\rangle, \Upsilon\langle b_1\psi_1, \ldots, b_n\psi_n\rangle)$. We show that $h\sigma \sim h\theta$.

   Denote $A = \Upsilon\langle a_1\theta_1, \ldots, a_n\theta_n\rangle$, $B = \langle b_1, \ldots, b_n\rangle$, and $\bar{B} = \Upsilon\langle b_1, \ldots, b_n\rangle$. Observe that there exists ψ such that $\Upsilon\langle b_1\psi_1, \ldots, b_n\psi_n\rangle = \bar{B}\psi$. In this notation $mgu(B, A) = \theta$, $mgu(B, \bar{B}\psi) = \sigma$ and by construction $\bar{B}\psi \sim \bar{B}\psi'$ where $\psi' = mgu(A, \bar{B})$. By Lemma 2 (below) it follows that $B\sigma \sim B\theta$, which implies (due to renaming) that $h\sigma \sim h\theta$.

2. We show that $[h\sigma]$ is an element of the right-hand-side of equation (1). Denote $\bar{\kappa}_i = mgu_g^A(b_i, a_i)$, $\hat{\kappa}_i = \sqcup\left\{ mgu_g^A(\langle b_i\rangle, \langle a\rangle) \;\middle|\; a \in heads(P) \right\}$, and $\kappa = mgu_{\hat{g}}^A(\bar{b}, \bar{b})$.

   By safety of abstract unification there exist $\psi_i' \in \bar{\gamma}(\bar{\kappa}_i)$ such that $b_i\psi_i' \sim b_i\psi_i$. By construction $\bar{\kappa}_i \sqsubseteq_{ASub} \hat{\kappa}_i$ and hence by monotonicity of $\bar{\gamma}$, $\psi_i' \in \bar{\gamma}(\hat{\kappa}_i)$.

   Safety of abstract unification implies that there exist $\sigma' \in \bar{\gamma}(\kappa)$ such that $\bar{b}\sigma \sim \bar{b}\sigma'$. □

LEMMA **2.** *Let A, B and $\bar{B}$ be syntactic objects such that $\bar{B}$ is less instantiated than B, $mgu(A, \bar{B}) = \psi'$, $mgu(A, B) = \theta$ and $mgu(B, \bar{B}\psi') = \sigma$. Then $B\sigma \sim B\theta$.*

PROOF. The proof relies on the following definitions (which are slightly non-standard: $\leq$ is not a partial order here but rather a pre-order). Let p, q and r be atoms. If $p \leq r$ and $q \leq r$ then we say that r is an upper bound for p and q; r is a least upper bound (lub) for p and q if r is an upper bound and for any other atom r' which is an upper bound of p and q, $r \leq r'$. It follows that if r and r' are both lubs of p and q then $r \sim r'$. If $\theta = mgu(p, q)$ then $p\theta$ is a lub for p and q. Assuming the premise of the lemma we have

1. (a) $\theta = mgu(A, B) \Rightarrow B\theta$ is a lub of A and B.
   (b) $\sigma = mgu(B, \bar{B}\psi') \Rightarrow B\sigma$ is a lub of B and $\bar{B}\psi'$.
   (c) $\psi' = mgu(A, \bar{B}) \Rightarrow \bar{B}\psi'$ is a lub of A and $\bar{B}$.
2. $\bar{B} \leq B$ by construction, so (1a) gives that $B\theta$ is an upper bound of A and $\bar{B}$.
3. From (1c) and (2) we get $\bar{B}\psi' \leq B\theta$ which implies that $B\theta$ is an upper bound for $\bar{B}\psi'$ and B.
4. $B\theta$ is also a lub for $\bar{B}\psi'$ and B because if C is any upper bound of $\bar{B}\psi'$ and B then $A \leq \bar{B}\psi' \leq C$ (from 1c) and $B \leq C$ so C is an upper bound of A and B. But $B\theta$ is a lub of A and B, so $B\theta \leq C$, which implies that $B\theta$ is a lub for $\bar{B}\psi'$ and B.
5. From (4) and (1b) we get that $B\theta$ and $B\sigma$ are both lubs for $\bar{B}\psi'$ and B; so $B\theta \sim B\sigma$.

□

THEOREM **6.7**
Consider a partial computation of a (sequential or pure) logic program P with initial goal G. Let $a \in atoms(P_G)$, $\varphi \in Sub$ and suppose that $a\varphi$ is a call in this computation. Then:

1. $$a^{\mathcal{C}}\varphi \in atoms([\![P_G^{\mathcal{M}}]\!]_{con}); \tag{2}$$
2. (a) if P is a sequential logic program and σ is an answer substitution for $a\varphi$, then
   $$\sigma \in ans_{[\![P_G^{\mathcal{M}}]\!]_{con}}(a^{\mathcal{A}}\varphi); \tag{3}$$
   (b) if P is a pure logic program and σ is a partial answer substitution for $a\varphi$, then
   $$\sigma \in ans_{[\![P_G^{\mathcal{M}}]\!]_{con}}(a^{\mathcal{P}}\varphi). \tag{4}$$

In the following, we abbreviate $atoms([\![P_G^{\mathcal{M}}]\!]_{con})$ and $ans_{[\![P_G^{\mathcal{M}}]\!]_{con}}(a)$ by *atoms* and $ans(a)$ respectively (for $a \in Atom$). Substitutions will always be assumed to be implicitly restricted, so we write $\sigma \in ans(a)$ rather than $\sigma \restriction vars(a) \in ans(a)$. When referring to a clause in P, it will be assumed to be an appropriate (depending on the context) renaming; similarly, we assume an appropriate renaming when referring to an element of *atoms*. An atom of the form $b(\_, \ldots, \_)$ will sometimes be written as $b(\_)$.

The following lemma is used to prove Theorem 6.7. It concerns a partial computation (of a program P_G) starting with the atomic goal $\langle a;\sigma\rangle$ (where $a \in atoms(P_G)$) where the first call, $a\sigma$, is reduced yielding a partial answer substitution ϑ. The lemma states that if we already know that the call $a\sigma$ is in the bottom-up meaning of the transformed program $P_G^{\mathcal{M}}$ (i.e., $a\sigma \in atoms$), then the partial answer substitution ϑ is also in the bottom-up meaning of $P_G^{\mathcal{M}}$ (i.e., $\vartheta \in ans(a^{\mathcal{P}}\sigma)$). A similar result holds for the direct subgoals of a. Observe that in the statement of the lemma $\vartheta = \theta\beta$.

LEMMA 3. *Let P_G be an initialized program, $a, b_1, \ldots, b_m \in atoms(P_G)$, $\bar{b}_1, \ldots, \bar{b}_m \in atoms(P_G)^*$ and $\sigma, \theta, \beta \in Sub$. Consider the following computation in P_G:*

$$\langle a;\sigma\rangle \stackrel{a\sigma}{\rightarrow} \langle b_1, \ldots, b_m; \sigma\theta\rangle \rightarrow^n \langle \bar{b}_1, \ldots, \bar{b}_m; \sigma\theta\beta\rangle \tag{5}$$

where $n \geq 0$ and $\beta = \beta_1 \cdots \beta_m$ such that for $1 \leq j \leq m$,

$$\langle b_j; \sigma\theta\beta_1 \cdots \beta_{j-1}\rangle \rightarrow^* \langle \bar{b}_j; \sigma\theta\beta_1 \cdots \beta_j\rangle. \tag{6}$$

Assume that

$$a^{\mathcal{C}}\sigma \in atoms \tag{7}$$

then:

$$\theta\beta \in ans(a^{\mathcal{P}}\sigma), \quad \text{and furthermore} \tag{8}$$
$$\beta_j \in ans(b_j^{\mathcal{P}}\sigma\theta\beta_1 \cdots \beta_{j-1}). \tag{9}$$

PROOF OF LEMMA 3. The proof is by induction on the lenght $n+1$ of the computation in (5). Note that (5) implies that there is a clause $h \leftarrow b_1, \ldots, b_m$ in P such that

$$\theta = mgu(a\sigma, h). \tag{10}$$

So by Definition 6.6, the following clauses are in $P_G^{\mathcal{M}}$ ($1 \leq j \leq m$):

$$b_j^{\mathcal{C}} \leftarrow h^{\mathcal{C}}, b_1^{\mathcal{P}}, \ldots, b_{j-1}^{\mathcal{P}}, b_{j+1}^{\mathcal{P}}, \ldots, b_m^{\mathcal{P}}. \tag{11}$$
$$b_j^{\mathcal{P}}(\_, \ldots, \_). \tag{12}$$
$$h^{\mathcal{P}} \leftarrow h^{\mathcal{C}}, b_1^{\mathcal{P}}, \ldots, b_m^{\mathcal{P}}. \tag{13}$$

Let $n = 0$. From (12) we know that $b_j^{\mathcal{P}}(\_) \in atoms$, so by the definition of $[\![P_G^{\mathcal{M}}]\!]_{con}$ (see Section 3.2) we have with (13), (7) and (10) that $h^{\mathcal{P}}\theta \in atoms$. (Operationally, we are unifying here the atoms in the body of clause (13) with $a^{\mathcal{C}}\sigma$ and the $b_j^{\mathcal{P}}(\_)$ respectively.) So, by (10), $a^{\mathcal{P}}\sigma\theta \in atoms$, and because $\beta = \epsilon$, $a^{\mathcal{P}}\sigma\theta\beta \in atoms$, from which (8) follows. (9) is also proven easily.

Now suppose $n > 0$. Unifying (see Table 2) the body of the clause $b_1^{\mathcal{C}} \leftarrow h^{\mathcal{C}}, b_2^{\mathcal{P}}, \ldots, b_m^{\mathcal{P}}$ for $j = 1$ in (11) with the atoms $a^{\mathcal{C}}\sigma, b_2^{\mathcal{P}}(\_), \ldots, b_m^{\mathcal{P}}(\_)$ obtained from (7) and from (12) by having j range from 2 to m gives, using (10):

$$b_1^{\mathcal{C}}\theta \in atoms \tag{14}$$

and because $dom(\sigma) \cap vars(b_1) = \emptyset$ (this follows from the form of (5)), (14) it follows that

$$b_1^{\mathcal{C}}\sigma\theta \in atoms \tag{15}$$

Because computation (6) for $j = 1$ has length smaller than $n+1$, we may apply the induction hypothesis to (15) to conclude:

$$\beta_1 \in ans(b_1^{\mathcal{P}}\sigma\theta). \tag{16}$$

clause body	$h^{\mathcal{C}},\ b_2^{\mathcal{P}},\ \ldots,\ b_m^{\mathcal{P}}$
old atoms	$a^{\mathcal{C}}\sigma, b_2^{\mathcal{P}}(\_), \ldots, b_m^{\mathcal{P}}(\_)$
new call	$b_1^{\mathcal{C}}\sigma\theta$
new answer	$\beta_1 \in ans(b_1^{\mathcal{P}}\sigma\theta)$

Table 2: Argument for $\beta_1 \in ans(b_1^{\mathcal{P}}\sigma\theta)$.

In a similar way as we derived (14), we can now unify (see Table 3) the body of the clause $b_2^{\mathcal{C}} \leftarrow h^{\mathcal{C}},\ b_1^{\mathcal{P}},\ b_3^{\mathcal{P}},\ \ldots,\ b_m^{\mathcal{P}}$ for $j = 2$ in (11) with the elements $a^{\mathcal{C}}\sigma, b_1^{\mathcal{P}}\sigma\theta\beta_1, b_3^{\mathcal{P}}(\_), \ldots, b_m^{\mathcal{P}}(\_)$ from *atoms* which are obtained from (7), from (16) and from (12) by having j range from 3 to m, giving (compare (15)):

$$b_2^{\mathcal{C}}\sigma\theta\beta_1 \in atoms. \tag{17}$$

The induction hypothesis implies again (compare (16)):

$$\beta_2 \in ans(b_2^{\mathcal{P}}\sigma\theta\beta_1). \tag{18}$$

clause body	$h^{\mathcal{C}},\ b_1^{\mathcal{P}},\ b_3^{\mathcal{P}},\ \ldots,\ b_m^{\mathcal{P}}$
old atoms	$a^{\mathcal{C}}\sigma, b_1^{\mathcal{P}}\sigma\theta\beta_1, b_3^{\mathcal{P}}(\_), \ldots, b_m^{\mathcal{P}}(\_)$
new call	$b_2^{\mathcal{C}}\sigma\theta\beta_1$
new answer	$\beta_2 \in ans(b_2^{\mathcal{P}}\sigma\theta\beta_1)$

Table 3: Argument for $\beta_2 \in ans(b_2^{\mathcal{P}}\sigma\theta\beta_1)$.

We can repeat this for the clauses for $j = 3$ up to m in (11) successively, to obtain in general for $1 \leq j \leq m$:

$$\beta_j \in ans(b_j^{\mathcal{P}}\sigma\theta\beta_1 \cdots \beta_{j-1}).$$

This completes the proof of (9). We now combine this result with (13) and (7) to unify the body of $h^{\mathcal{P}} \leftarrow h^{\mathcal{C}},\ b_1^{\mathcal{P}},\ \ldots,\ b_m^{\mathcal{P}}$ with the elements $a^{\mathcal{C}}\sigma, b_1^{\mathcal{P}}\sigma\theta\beta_1, \ldots, b_m^{\mathcal{P}}\sigma\theta\beta_1 \cdots \beta_m$ of *atoms* giving (see Table 4)

$$h^{\mathcal{P}}\sigma\theta\beta_1 \cdots \beta_m \in atoms. \tag{19}$$

With (10) this now implies (recall that $\beta_1 \cdots \beta_m = \beta$)

$$\theta\beta = mgu(a^{\mathcal{P}}\sigma, h^{\mathcal{P}}\sigma\theta\beta). \tag{20}$$

Thus,

$$\theta\beta \in ans(a^{\mathcal{P}}\sigma)$$

clause body	$h^{\mathcal{C}}, \quad b_1^{\mathcal{P}}, \quad \ldots, \quad b_m^{\mathcal{P}}$
old atoms	$a^{\mathcal{C}}\sigma, b_1^{\mathcal{P}}\sigma\theta\beta_1, \ldots, b_m^{\mathcal{P}}\sigma\theta\beta_1 \cdots \beta_m$
new call	$h^{\mathcal{P}}\sigma\theta\beta_1 \cdots \beta_m$
new answer	$\theta\beta \in ans(a^{\mathcal{P}}\sigma)$

Table 4: Argument for $\theta\beta \in ans(a^{\mathcal{P}}\sigma)$.

which completes the proof of (8). □

We now proceed with the proof of Theorem 6.7. We prove 1 and 2(b); the proof of 2(a) is similar.

PROOF OF THEOREM **6.7.**

1. The proof of (2) is by induction on the depth d of (an occurence of) $a\varphi$ in the computation tree which is induced by the transition relation $\rightarrow_P$. We assume without loss of generality that the initial goal is atomic; its depth is 0.

   base: $d = 0$:
   There is one call with depth 0, which is $G\epsilon$. Because $G^{\mathcal{C}}$ is a fact in $P_G^{\mathcal{M}}$, $G^{\mathcal{C}}\epsilon \in atoms$.

   induction step: $d + 1$:
   Consider the call $b_i\varphi\theta\rho$ of depth $d + 1$ in the following computation:

$$\langle G; \epsilon\rangle \rightarrow^* \langle \bar{g}_1, a, \bar{g}_2; \varphi\rangle \stackrel{a\varphi}{\rightarrow} \langle \bar{g}_1, b_1, \ldots, b_m, \bar{g}_2; \varphi\theta\rangle \rightarrow^* \langle \bar{g}_1', \ldots, b_i, \ldots, \bar{g}_2'; \varphi\theta\rho\rangle \stackrel{b_i\varphi\theta\rho}{\rightarrow} \tag{21}$$

   in which $\bar{g}_j$ and $\bar{g}_j'$ $(j = 1, 2)$ are conjunctive goals. In [12], Falaschi and Levi prove a generalization of the Switching Lemma [17] which applies to the case of partial computations. Specialized to computation (21) above, it states that when we change the order in which the atoms are selected for reduction, the last state of the resulting computation will be a variant (renaming) of the last state of (21). Therefore we may assume without loss of generality that after the selection of atom a in the computation, no more reduction of atoms in the $\bar{g}_j$ take place, in other words, we may assume that $\bar{g}_j' = \bar{g}_j$ $(j = 1, 2)$. By the same lemma we may assume that the reductions of $b_1, \ldots, b_{i-1}, b_{i+1}, \ldots, b_m$ take place in a left-to-right fashion. So the computation has the following form:

$$\langle G; \epsilon\rangle \rightarrow^* \langle \bar{g}_1, a, \bar{g}_2; \varphi\rangle \stackrel{a\varphi}{\rightarrow} \langle \bar{g}_1, b_1, \ldots, b_m, \bar{g}_2; \varphi\theta\rangle \rightarrow^* \langle \bar{g}_1, \bar{b}_1, \ldots, \bar{b}_{i-1}, b_i, \bar{b}_{i+1}, \ldots, \bar{b}_m, \bar{g}_2; \varphi\theta\rho\rangle \stackrel{b_i\varphi\theta\rho}{\rightarrow} \tag{22}$$

and there exist $\beta_1, \ldots, \beta_{i-1}, \beta_{i+1}, \ldots, \beta_m$ such that $\rho = \beta_1 \cdots \beta_{i-1}\beta_{i+1} \cdots \beta_m$ and:

$$\text{for } 1 \leq j \leq i-1: \quad \langle b_j; \varphi\theta\beta_1 \cdots \beta_{j-1} \rangle \rightarrow^* \langle \bar{b}_j; \varphi\theta\beta_1 \cdots \beta_{j-1}\beta_j \rangle \tag{23}$$

$$\text{for } i+1 \leq j \leq m: \quad \langle b_j; \varphi\theta\beta_1 \cdots \beta_{i-1}\beta_{i+1} \cdots \beta_{j-1} \rangle \rightarrow^* \langle \bar{b}_j; \varphi\theta\beta_1 \cdots \beta_{i-1}\beta_{i+1} \cdots \beta_{j-1}\beta_j \rangle. \tag{24}$$

Because the call $a\varphi$ has depth $\leq d$, we may apply the induction hypothesis to derive

$$a^{\mathcal{C}}\varphi \in \mathit{atoms}. \tag{25}$$

We can now apply Lemma 3 (see (9)) to this to infer

$$\text{for } 1 \leq j \leq i-1: \quad \beta_j \in \mathit{ans}(b_j^{\mathcal{P}}\varphi\theta\beta_1 \cdots \beta_{j-1})$$

$$\text{for } i+1 \leq j \leq m: \quad \beta_j \in \mathit{ans}(b_j^{\mathcal{P}}\varphi\theta\beta_1 \cdots \beta_{i-1}\beta_{i+1} \cdots \beta_{j-1}). \tag{26}$$

We know there is a clause $h \leftarrow b_1, \ldots, b_m$ in P, so in $P_G^{\mathcal{M}}$ there is the clause:

$$b_i^{\mathcal{C}} \leftarrow h^{\mathcal{C}}, b_1^{\mathcal{P}}, \ldots, b_{i-1}^{\mathcal{P}}, b_{i+1}^{\mathcal{P}}, \ldots, b_m^{\mathcal{P}} \tag{27}$$

such that

$$\theta = \mathit{mgu}(a\varphi, h). \tag{28}$$

Now unifying the atoms in the body of clause (27) with the atoms obtained from (25) and (26) gives

$$b_i^{\mathcal{C}}\varphi\theta\beta_1 \cdots \beta_{i-1}\beta_{i+1} \cdots \beta_m \in \mathit{atoms}$$

which completes the proof of (2).

2. Consider the partial answer substitution $\theta\beta$ to the call $a\varphi$ in the following computation:

$$\langle G; \epsilon \rangle \rightarrow^* \langle \bar{g}_1, a, \bar{g}_2; \varphi \rangle \stackrel{a\varphi}{\rightarrow} \langle \bar{g}_1, b_1, \ldots, b_m, \bar{g}_2; \varphi\theta \rangle \rightarrow^* \langle \bar{g}_1, \bar{b}_1, \ldots, \bar{b}_m, \bar{g}_2; \varphi\theta\beta \rangle \tag{29}$$

(again, by the generalized Switching Lemma we may assume that the $\bar{g}_i$ do not change). From the form of this computation, it follows there is the following clause in $P_G^{\mathcal{M}}$:

$$h^{\mathcal{P}} \leftarrow h^{\mathcal{C}}, b_1^{\mathcal{P}}, \ldots, b_m^{\mathcal{P}}. \tag{30}$$

From the first part of the theorem, (2), we already know that

$$a^{\mathcal{C}}\varphi \in \mathit{atoms}. \tag{31}$$

The generalized Switching Lemma again implies that we can assume without loss of generality that the atoms b_j $(1 \leq j \leq m)$ in (29) are solved in a left-to-right order, so that we can apply Lemma 3, giving

$$\beta_j \in \mathit{ans}(b_j^{\mathcal{P}}\varphi\theta\beta_1 \cdots \beta_{j-1}) \tag{32}$$

Now (30), (31) and (32) give

$$\theta\beta \in \mathit{ans}(A^{\mathcal{P}}\varphi).$$

References

[1] F. Bancilhon, D. Maier, Y. Sagiv, and J. Ullman. Magic Sets and other strange ways to implement logic programs. In *Proceedings of the 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, pages 1–15, 1986.

[2] R. Barbuti, M. Codish, R. Giacobazzi, and G. Levi. Modelling Prolog control. In *Proceedings of the 19th ACM Symposium on the Principles of Programming Languages*, pages 95–104. ACM Press, 1992. Submitted for publication.

[3] R. Barbuti, R. Giacobazzi, and G. Levi. A declarative approach to abstract interpretation of logic programs. Technical Report TR-20/89, Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy, 1989.

[4] R. Barbuti, R. Giacobazzi, and G. Levi. A general framework for semantics-based bottom-up abstract interpretation of logic programs. Technical Report TR 12/91, Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy, 1991. To appear in *ACM Transactions on Programming Languages and Systems.*

[5] C. Beeri and R. Ramakrishnan. On the power of magic. In *Proceedings of the 6th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, San Diego, California, 1987.

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

[7] F. Bry. Query evaluation in recursive databases: Bottom-up and top-down reconciled. Technical report, ECRC, Munich, 1989. An shorter version of this paper appeared at the 1st International Conference on Deductive and Object-Oriented Databases, Kyoto, Japan, 1989.

[8] M. Codish, D. Dams, and E. Yardeni. Abstract unification and a bottom-up analysis to detect aliasing in logic programs. Technical Report CS90-10, Weizmann Institute of Science, Department of Computer Science, May 1990.

[9] M. Codish, M. Falaschi, and K. Marriott. Suspension analysis for concurrent logic programs. In K. Furukawa, editor, *Proceedings of the 8th International Conference on Logic Programming*, pages 331–345. The MIT Press, Cambridge, Massachusetts, 1991. Submitted for publication.

[10] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on the Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977.

[11] S. Debray and R. Ramakrishnan. Canonical computations of logic programs. Technical report, Department of Computer Science, University of Arizona-Tucson, July 1990. Submitted for publication.

[12] M. Falaschi and G. Levi. Finite failures and partial computations in concurrent logic languages. *Theoretical Computer Science*, 75:45–66, 1990.

[13] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modeling of the operational behavior of logic languages. *Theoretical Computer Science*, 69(3):289–318, 1989.

[14] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A model-theoretic reconstruction of the operational semantics of logic programs. Technical Report TR-32/89, Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy, June 1989. To appear in *Information and Computation*.

[15] N. C. Heintze and J. Jaffar. A finite presentation theorem for approximating logic programs. In *Proceedings of the 17^{th} ACM Symposium on the Principles of Programming Languages*, pages 197–209, 1990.

[16] R. Kemp and G. Ringwood. An algebraic framework for the abstract interpretation of logic programs. In S. Debray and M. Hermenegildo, editors, *Proceedings of the North American Conference on Logic Programming*, pages 506–520. The MIT Press, Cambridge, Massachusetts, 1990.

[17] J.W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2^{nd} edition, 1987.

[18] K. Marriott and H. Søndergaard. Bottom-up abstract interpretation of logic programs. In *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, Washington, Seattle, August 1988.

[19] K. Marriott and H. Søndergaard. Semantics-based dataflow analysis of logic programs. In G. Ritter, editor, *Information Processing 89*. North-Holland, 1989.

[20] C. Mellish. Abstract interpretation of prolog programs. In S. Abramsky and C. Hankin, editors, *Abstract Interpretation of Declarative Languages*, pages 181–198. Ellis Horwood Ltd, 1987.

[21] A. Melton, D. Schmidt, and G. Strecker. Galois connections and computer science applications. In D. Pitt *et al.*, editor, *Category Theory and Computer Programming*, pages 299–312. Springer-Verlag, 1986. Lecture Notes in Computer Science 240.

[22] K. Muthukumar and M. Hermenegildo. Determination of variable dependence information through abstract interpretation. In *Proceedings of the North American Conference on Logic Programming*, Cleveland, Ohio, October 1989. MIT Press. To appear in the Journal of Logic Programming.

[23] F. Nielson. Strictness analysis and denotational abstract interpretation. *Information and Computation*, 76:29–92, 1988.

[24] U. Nilsson. Abstract interpretation: A kind of magic. In *Programming Language Implementation and Logic Programming 91*, pages 299–309. Springer-Verlag, 1991. Extended version to appear in the Journal of Theoretical Computer Science.

[25] R. Ramakrishnan. Magic templates: A spellbinding approach to logic programs. In *Proceedings of the 5^{th} International Conference and Symposium on Logic Programming*, August 1988.

[26] R. Ramakrishnan. Magic templates: A spellbinding approach to logic programs. *Journal of Logic Programming*, 11:189–216, 1991.

[27] J. Rohmer, R. Lescœr, and J.-M. Kerisit. The Alexander method, a technique for the processing of recusive axioms in deductive databases. *New Generation Computing*, 4, 1986.

[28] H. Schmidt, W. Kiessling, U. Güntzer, and R. Bayer. Compiling exploratory and goal-directed deduction into sloppy delta-iteration. In *Proceedings of the Symposium on Logic Programming*, pages 234–243, San Francisco, California, 1987. Computer Society Press of IEEE.

[29] H. Seki. On the power of Alexander templates. In *Proceedings of the 8^{th} ACM SIGACT-SIGMOD-SIGART Symposium on the Principles of Database Systems*, Philadelphia, Pennsylvania, 1989.

[30] H. Søndergaard. Semantic based analysis and transformation of logic programs. Technical Report 12, The University of Melbourne, June 1990. Revised version of PhD thesis, University of Copenhagen, December 1989.

[31] J. Steiner. Personal communication.

[32] H. Tamaki and T. Sato. OLD resolution with tabulation. In E. Y. Shapiro, editor, *Proceedings of the 3^{rd} International Conference on Logic Programming*, London, July 1986. Springer-Verlag. Lecture Notes in Computer Science Vol. 225.

[33] E. Yardeni and E. Shapiro. A type system for logic programs. *Journal of Logic Programming*, 10:125–135, 1991.

In this series appeared:

90/1	W.P.de Roever- H.Barringer- C.Courcoubetis-D.Gabbay R.Gerth-B.Jonsson-A.Pnueli M.Reed-J.Sifakis-J.Vytopil P.Wolper	Formal methods and tools for the development of distributed and real time systems, p. 17.
90/2	K.M. van Hee P.M.P. Rambags	Dynamic process creation in high-level Petri nets, pp. 19.
90/3	R. Gerth	Foundations of Compositional Program Refinement - safety properties - , p. 38.
90/4	A. Peeters	Decomposition of delay-insensitive circuits, p. 25.
90/5	J.A. Brzozowski J.C. Ebergen	On the delay-sensitivity of gate networks, p. 23.
90/6	A.J.J.M. Marcelis	Typed inference systems : a reference document, p. 17.
90/7	A.J.J.M. Marcelis	A logic for one-pass, one-attributed grammars, p. 14.
90/8	M.B. Josephs	Receptive Process Theory, p. 16.
90/9	A.T.M. Aerts P.M.E. De Bra K.M. van Hee	Combining the functional and the relational model, p. 15.
90/10	M.J. van Diepen K.M. van Hee	A formal semantics for Z and the link between Z and the relational algebra, p. 30. (Revised version of CSNotes 89/17).
90/11	P. America F.S. de Boer	A proof system for process creation, p. 84.
90/12	P.America F.S. de Boer	A proof theory for a sequential version of POOL, p. 110.
90/13	K.R. Apt F.S. de Boer E.R. Olderog	Proving termination of Parallel Programs, p. 7.
90/14	F.S. de Boer	A proof system for the language POOL, p. 70.
90/15	F.S. de Boer	Compositionality in the temporal logic of concurrent systems, p. 17.
90/16	F.S. de Boer C. Palamidessi	A fully abstract model for concurrent logic languages, p. p. 23.
90/17	F.S. de Boer C. Palamidessi	On the asynchronous nature of communication in logic languages: a fully abstract model based on sequences, p. 29.

90/18	J.Coenen E.v.d.Sluis E.v.d.Velden	Design and implementation aspects of remote procedure calls, p. 15.
90/19	M.M. de Brouwer P.A.C. Verkoulen	Two Case Studies in ExSpect, p. 24.
90/20	M.Rem	The Nature of Delay-Insensitive Computing, p.18.
90/21	K.M. van Hee P.A.C. Verkoulen	Data, Process and Behaviour Modelling in an integrated specification framework, p. 37.
91/01	D. Alstein	Dynamic Reconfiguration in Distributed Hard Real-Time Systems, p. 14.
91/02	R.P. Nederpelt H.C.M. de Swart	Implication. A survey of the different logical analyses "if...,then...", p. 26.
91/03	J.P. Katoen L.A.M. Schoenmakers	Parallel Programs for the Recognition of *P*-invariant Segments, p. 16.
91/04	E. v.d. Sluis A.F. v.d. Stappen	Performance Analysis of VLSI Programs, p. 31.
91/05	D. de Reus	An Implementation Model for GOOD, p. 18.
91/06	K.M. van Hee	SPECIFICATIEMETHODEN, een overzicht, p. 20.
91/07	E.Poll	CPO-models for second order lambda calculus with recursive types and subtyping, p. 49.
91/08	H. Schepers	Terminology and Paradigms for Fault Tolerance, p. 25.
91/09	W.M.P.v.d.Aalst	Interval Timed Petri Nets and their analysis, p.53.
91/10	R.C.Backhouse P.J. de Bruin P. Hoogendijk G. Malcolm E. Voermans J. v.d. Woude	POLYNOMIAL RELATORS, p. 52.
91/11	R.C. Backhouse P.J. de Bruin G.Malcolm E.Voermans J. van der Woude	Relational Catamorphism, p. 31.
91/12	E. van der Sluis	A parallel local search algorithm for the travelling salesman problem, p. 12.
91/13	F. Rietman	A note on Extensionality, p. 21.
91/14	P. Lemmens	The PDB Hypermedia Package. Why and how it was built, p. 63.

91/15	A.T.M. Aerts K.M. van Hee	Eldorado: Architecture of a Functional Database Management System, p. 19.
91/16	A.J.J.M. Marcelis	An example of proving attribute grammars correct: the representation of arithmetical expressions by DAGs, p. 25.
91/17	A.T.M. Aerts P.M.E. de Bra K.M. van Hee	Transforming Functional Database Schemes to Relational Representations, p. 21.
91/18	Rik van Geldrop	Transformational Query Solving, p. 35.
91/19	Erik Poll	Some categorical properties for a model for second order lambda calculus with subtyping, p. 21.
91/20	A.E. Eiben R.V. Schuwer	Knowledge Base Systems, a Formal Model, p. 21.
91/21	J. Coenen W.-P. de Roever J.Zwiers	Assertional Data Reification Proofs: Survey and Perspective, p. 18.
91/22	G. Wolf	Schedule Management: an Object Oriented Approach, p. 26.
91/23	K.M. van Hee L.J. Somers M. Voorhoeve	Z and high level Petri nets, p. 16.
91/24	A.T.M. Aerts D. de Reus	Formal semantics for BRM with examples, p. 25.
91/25	P. Zhou J. Hooman R. Kuiper	A compositional proof system for real-time systems based on explicit clock temporal logic: soundness and complete ness, p. 52.
91/26	P. de Bra G.J. Houben J. Paredaens	The GOOD based hypertext reference model, p. 12.
91/27	F. de Boer C. Palamidessi	Embedding as a tool for language comparison: On the CSP hierarchy, p. 17.
91/28	F. de Boer	A compositional proof system for dynamic proces creation, p. 24.
91/29	H. Ten Eikelder R. van Geldrop	Correctness of Acceptor Schemes for Regular Languages, p. 31.
91/30	J.C.M. Baeten F.W. Vaandrager	An Algebra for Process Creation, p. 29.
91/31	H. ten Eikelder	Some algorithms to decide the equivalence of recursive types, p. 26.

91/32	P. Struik	Techniques for designing efficient parallel programs, p. 14.
91/33	W. v.d. Aalst	The modelling and analysis of queueing systems with QNM-ExSpect, p. 23.
91/34	J. Coenen	Specifying fault tolerant programs in deontic logic, p. 15.
91/35	F.S. de Boer J.W. Klop C. Palamidessi	Asynchronous communication in process algebra, p. 20.
92/01	J. Coenen J. Zwiers W.-P. de Roever	A note on compositional refinement, p. 27.
92/02	J. Coenen J. Hooman	A compositional semantics for fault tolerant real-time systems, p. 18.
92/03	J.C.M. Baeten J.A. Bergstra	Real space process algebra, p. 42.
92/04	J.P.H.W.v.d.Eijnde	Program derivation in acyclic graphs and related problems, p. 90.
92/05	J.P.H.W.v.d.Eijnde	Conservative fixpoint functions on a graph, p. 25.
92/06	J.C.M. Baeten J.A. Bergstra	Discrete time process algebra, p.45.
92/07	R.P. Nederpelt	The fine-structure of lambda calculus, p. 110.
92/08	R.P. Nederpelt F. Kamareddine	On stepwise explicit substitution, p. 30.
92/09	R.C. Backhouse	Calculating the Warshall/Floyd path algorithm, p. 14.
92/10	P.M.P. Rambags	Composition and decomposition in a CPN model, p. 55.
92/11	R.C. Backhouse J.S.C.P.v.d.Woude	Demonic operators and monotype factors, p. 29.
92/12	F. Kamareddine	Set theory and nominalisation, Part I, p.26.
92/13	F. Kamareddine	Set theory and nominalisation, Part II, p.22.
92/14	J.C.M. Baeten	The total order assumption, p. 10.
92/15	F. Kamareddine	A system at the cross-roads of functional and logic programming, p.36.
92/16	R.R. Seljée	Integrity checking in deductive databases; an exposition, p.32.
92/17	W.M.P. van der Aalst	Interval timed coloured Petri nets and their analysis, p. 20.

92/18	R.Nederpelt F. Kamareddine	A unified approach to Type Theory through a refined lambda-calculus, p. 30.
92/19	J.C.M.Baeten J.A.Bergstra S.A.Smolka	Axiomatizing Probabilistic Processes: ACP with Generative Probabilities, p. 36.
92/20	F.Kamareddine	Are Types for Natural Language? P. 32.
92/21	F.Kamareddine	Non well-foundedness and type freeness can unify the interpretation of functional application, p. 16.
92/22	R. Nederpelt F.Kamareddine	A useful lambda notation, p. 17.
92/23	F.Kamareddine E.Klein	Nominalization, Predication and Type Containment, p. 40.
92/24	M.Codish D.Dams Eyal Yardeni	Bottum-up Abstract Interpretation of Logic Programs, p. 33.
92/25	E.Poll	A Programming Logic for Fω, p. 15.