Detecting Determinate Computations by Bottom-up Abstract Interpretation *

Roberto Giacobazzi , Laura Ricci
Dipartimento di Informatica
Università di Pisa
Corso Italia 40, 56125 Pisa
{giaco,ricci}@di.unipi.it

Abstract

One of the most interesting characteristics of logic programs is the ability of expressing nondeterminism in an elegant and concise way. On the other hand, implementation of nondeterminism poses serious problems both in the sequential and in the concurrent case. If determinate computations are detected through a static analysis, a noticeable reduction of the execution time may be obtained. This work describes a static analysis to detect determinate computations. The analysis does not require the knowledge of the activating modes of the predicate and it derives determinacies which may be inferred from the success set of the program.

1 Introduction

One of the main features of logic programming is the ability to compute a set of output bindings for each variable of a query. While this characteristic supports an elegant and concise definition of non-deterministic behaviours, it poses serious problems both in a sequential and in a concurrent implementation of the language. In several cases, the ability of logic programs to compute multiple bindings is not exploited: in this case a logic program has a determinate behaviour, i.e. it produces a single output value for a given query, or for a class of queries satisfying some properties. Recently, some analyses [6, 4, 11, 10, 12] have been proposed to statically detect when a computation is determinate. The knowledge of determinate computations, i.e. determinacies, supports optimization of the program execution time [5] end of bottom-up evaluators [10].

Even if the analysis to detect determinancies is related to the control features of the language implementation, we show that determinancies may be inferred by a bottom up analysis that does not require any knowledge of the type and activating modes of a predicate. The analysis derives those determinacies that may be inferred from the success set of the program. The definition of the analysis requires the investigation of the relation between the input values of a predicate and those returned by the refutation of the

*To appear in *European Conference on Programming* ESOP'92

predicate itself. Some recent proposals [1, 6, 8, 12] characterize such a relation through the notion of dependence. This notion has been exploited, for instance, to describe the relation between the groundness of a set of arguments of a predicate upon invocation and the groundness of another set of arguments, after the refutation of the predicate. We refine such notion through that of *deterministic ground dependence*. A ground dependence from a set of input arguments to a set of output ones is deterministic if, whenever the input arguments are ground, the refutation binds any output argument to a single ground term. The definition of deterministic ground dependences supports a refined notion of functionality, i.e. functionality is inferred with respect to subsets of arguments rather than to whole predicates. The analysis has been defined and validated through a bottom-up abstract interpretation technique. It integrates the theory of hypergraphs with the semantics of logic programs in order to obtain a powerful program analysis tool. The information useful to characterize deterministic computations is produced by applying an abstract interpretation which returns an abstract model of the program. This approximation is obtained by the finite computation of the least fixpoint of an abstraction of the immediate consequence operator T_P. In order to have such an abstraction, a suitable set of abstract domains and operators, supporting the definition of T_P, is given.

Section 2 introduces some basic notions on hypergraph theory and semantics of logic programs, Section 3 formally defines the abstract domains and abstract operators for the analysis and abstract fixpoint semantics for deterministic dependences analysis, Section 4 shows some significant examples, while Section 5 deals with some concluding remark.

2 Preliminaries

We assume the reader familiar with the notions of lattice theory, hypergraph theory [2], semantics of logic programs [7] and the basic concepts of logic programming [9].

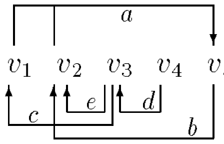
In the following we will denote by $S/\{s\}$ the set S where the element $\{s\}$ has been removed.

A *hypergraph* $\mathcal{G}$ is a pair $(\mathcal{N}, \mathcal{E})$ where $\mathcal{N} = \{v_1, ..., v_n\}$ is a (finite) set of *nodes* and $\mathcal{E} = \{e_1, ..., e_m\}$ is a set of *hyperarcs*, where for each $i = 1, ..., m$, $e_i = \langle T, H \rangle$ such that $T \subseteq \mathcal{N}$ (the tail) and $H \in \mathcal{N}$ (the head). In the following we denote by $Tail(e)$ and $Head(e)$ the tail and the head of any hyperarc e. The "tail" and "head" notions are extended to deal with sets of hyperarcs. A hyperarc e is *satisfied* by a set of nodes $\mathcal{N}'$ iff $Tail(e) \subseteq \mathcal{N}'$. A *graph* Ψ is a hypergraph $(\mathcal{N}_\Psi, \mathcal{E}_\Psi)$ where $\forall e \in \mathcal{E}_\Psi,\ |Tail(e)| = 1$ (we denote by $|T|$ the number of elements is T). Given a graph Ψ, the nodes n and n' are *connected in* Ψ iff there exists a path $\pi \subseteq \mathcal{E}_\Psi$ where $\pi = \{e_1, ..., e_k\}$ such that $n = Tail(e_1),\ n' = Head(e_k)$ and $\forall j = 1, ..., k-1,\ Head(e_j) = Tail(e_{j+1})$. The connection between the nodes n and n' (by means of a path π) in a graph Ψ will be denoted by: $n \mapsto_\Psi^\pi n'$. The graph, denoted by $\Psi_\mathcal{G}$, associated with the hypergraph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$ is obtained by reducing each hyperarc: $\langle \{s_1, ..., s_m\}, h \rangle \in \mathcal{E}$, in a set of arcs $\{\langle s_1, h \rangle ..., \langle s_m, h \rangle\}$. In the following we will denote by $\Im_\mathcal{G}(n', n) = \{\pi \mid n' \mapsto_{\Psi_\mathcal{G}}^\pi n\}$ the set of paths π defining a connection from n' to n in the graph $\Psi_\mathcal{G}$ associated with the hypergraph $\mathcal{G}$.

Given a hypergraph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, let $\mathcal{E}' \subseteq \mathcal{E}$. The sets of nodes $\mathcal{N}'$ and the node n are *connected* in $\mathcal{G}$, by means of the hyperarcs in $\mathcal{E}'$, denoted by $\mathcal{N}' \to_\mathcal{G}^{\mathcal{E}'} n$, iff all the hyperarcs in $\mathcal{E}'$ are satisfied by $\mathcal{N}' \cup Head(\mathcal{E}')$, $\forall i \in \mathcal{N}'$: $\Im_{(\mathcal{N}, \mathcal{E}')}(i, n) \neq \emptyset$ and

$\forall e \in \mathcal{E}', \ \forall i \in Tail(e) : \ i \notin \mathcal{N}', \ \exists \bar{n} \in \mathcal{N}'$ such that $\Im_{(\mathcal{N},\mathcal{E}'/\{e\})}(\bar{n}, i) \neq \emptyset$. We call $\mathcal{E}'$ a *connection* from $\mathcal{N}'$ to n in $\mathcal{G}$. A connection $\mathcal{E}'$ from $\mathcal{N}'$ to n in a hypergraph $\mathcal{G}$ is *minimal* iff for each $e \in \mathcal{E}$, $\mathcal{N}' \not\to_{\mathcal{G}}^{\mathcal{E}'/\{e\}} n$.

Example 2.1 *Let us consider the following hypergraph and some connections with* v_5

$\mathcal{G} =$ 

$$\{v_4\} \to_{\mathcal{G}}^{\{d,c,e,a\}} v_5, \ \{v_3\} \to_{\mathcal{G}}^{\{c,e,a\}} v_5, \ \{v_1, v_2\} \to_{\mathcal{G}}^{\{a\}} v_5, \ \ldots$$

while $\forall \mathcal{E}' \subseteq \{a, b, c, d, e\}$*:* $\{v_1\} \not\to_{\mathcal{G}}^{\mathcal{E}'} v_5$, $\{v_5\} \not\to_{\mathcal{G}}^{\mathcal{E}'} v_5$.

Notice that the definition of connection requires that any node in the tail of an hyperarc which is not in the set of starting nodes $\mathcal{N}'$ *can be reached from some starting node, without using the hyperarc itself. This condition avoids considering, in a connection, a hyperarc (e.g. a), such that each non-starting node* (v_2) *in the tail can only be reached from some starting node* (v_1) *by means of a path using the hyperarc itself (the hyperarc becomes auto-satisfied). As a consequence* $\not\exists \mathcal{E}' \subseteq \{a, b, c, d, e\} : \ \{v_1\} \to_{\mathcal{G}}^{\mathcal{E}'} v_5$.

The declarative semantics of logic programs in [7] is based on an *extended Herbrand Universe* containing also non-ground terms. It allows to declaratively characterize the ability of logic programs to compute answer substitutions (which are non-ground in general). Let us consider the set $Cons$ of term constructors, and a denumerable set Var of variables. The free $Cons$-algebra on Var is denoted as $T_{Cons}(Var)$. A substitution ϑ is a mapping from Var into $T_{Cons}(Var)$, such that $\{x \in Var \mid \vartheta(x) \neq x\}$ is finite. ε denotes the empty substitution. $\vartheta_{|G}$ denotes the restriction of the substitution ϑ to the variables occurring in G, which is extended as an identity, for any $x \in Var(G)$ such that $\vartheta(x)$ is undefined. We denote by mgu a total function which maps a pair of syntactic objects (e.g. atoms, terms,...) to an idempotent most general unifier of the objects, if such exists; $fail$ otherwise. The *extended Herbrand Universe* U_P is defined as $T_{Cons_P}(Var)/\approx$, where $t_1 \approx t_2$ iff $\exists \ \vartheta_1, \vartheta_2 \mid t_1\vartheta_1 = t_2 \ \wedge \ t_2\vartheta_2 = t_1$. Let $Pred$ be a finite set of predicate symbols. An *atom* is an object of the form $p(t_1, ..., t_n)$ where $t_1, ..., t_n \in T_{Cons_P}(Var)$ and p is an n-ary predicate symbol (i.e. $p \in Pred^n$). The set of atoms is denoted $Atoms$. We denote by $Var(a)$ the set of variables in any syntactic object a. The *Base of interpretations* B_P is $Atoms/_{\approx}$ where $\approx$ extends to atoms in the obvious way. An *interpretation* I is any subset of B_P. Standard results on model-theoretic and fixpoint semantics apply to the extended domains as well as in the ground case [7]. In particular, the fixpoint semantics for a logic program P (denoted as $\mathcal{F}(P)$) is defined by means of an immediate consequences operator T_P [7]. It derives possibly non-ground atoms by means of a bottom-up inference rule which is based on unification, as in the top-down SLD resolution.

We assume the reader familiar with the basic notions of *abstract interpretation* as defined in [3]. In the following an abstract interpretation supporting our analysis for definite logic programs is developed according to a bottom-up technique [1] based on an abstraction of the declarative and fixpoint semantics in [7].

3 Abstract Domains

This section introduces an analysis for ground and ground-deterministic dependences. Informally, a ground dependence does exist between a set of arguments T of an atom

A and another argument h, iff whenever the arguments in T are ground, the refutation of A produces a set of ground bindings for h. A ground dependence is determinate, i.e. deterministic ground dependence, iff the refutation produces at most one binding for h. Consider the logic program

$$append([], X, X) :-.$$

$$append([X|Xs], Y, [X|Ys]) :- append(Xs, Y, Ys).$$

the refutation of the goal $append([a|[]], [b|[]], Z)$ in the program produces only one binding for Z ($\{Z = [a|b|[]]\}$), while the goal $append(X, Y, [a|b|[]])$ produces multiple bindings for X and Y ($\{X = [a|[]], Y = [b|[]]\}$, $\{X = [a|b|[]], Y = []\}$, etc.). In the first case the groundness of the first two arguments uniquely determines the groundness of the third one, while, in the second case, a non-deterministic dependence does exist between the third argument and the first one and between the third and the second one.

Let P be a definite logic program, we denote by $U_P^\sharp = \{g, ng\}$ the *abstract universe of terms* where g represents the ground term set, and ng the possibly non-ground one. Two terms are ground-equivalent ($\approx_{ground}$) iff both are ground or non-ground terms. As a consequence, $U_P^\sharp$ can be considered as a set of abstract objects representing equivalence classes of terms, modulo ground equivalency. It is trivial to prove that $U_P^\sharp$ is a finite lattice since for each ground term there exists a non-ground term which is greater than it (i.e. $g \leq_g ng$). The elements in $U_P^\sharp$ will be denoted as $t^\sharp$.

Definition 3.1 *A* dependence representation $\mathcal{D}$ *is a triple* $(\mathcal{N}, \mathcal{V}, \mathcal{E})$, *where*

- $(\mathcal{N}, \mathcal{E})$ *is a hypergraph such that*

  $\mathcal{N}$ *is a finite set of symbols called the* domain *of the representation* ($Dom(\mathcal{D})$),

  $\mathcal{E}$ *is a set of labelled hyperarcs of the form* $\langle T, h\rangle_l$, *where* $l \in \{d, ?, \perp\}$,

- $\mathcal{V} \subseteq \mathcal{N} \times U_P^\sharp$ *such that* $|\mathcal{V}| = |\mathcal{N}| \ \wedge \ \forall (s, t^\sharp), (s', t'^\sharp) \in \mathcal{V}, \ s \neq s'$.

A *dependence* is a hyperarc in a dependence representation. An *autodependence* is a dependence $\langle T, h\rangle_l$ such that $h \in T$. Dependence representations which are defined over the same domain $\mathcal{N}$ are called *homogeneous*.

The hypergraph notation is useful to represent dependence information in a concise way. Dependence representations are hypergraphs enhanced with a suitable set of labels (for nodes and hyperarcs). Since both the elements of $\mathcal{V}$ and the elements of $\mathcal{E}$ are defined on the same set of nodes $\mathcal{N}$, we will often omit the reference to the underlying set of nodes when this can be deduced by the context. Labels associated with nodes will correspond to the *mode information* (a mode analysis is considered in our framework), while the labels associated with the dependence notion represent the type of the corresponding dependence (d and ? mean *deterministic* and *don't know* respectively, while $\perp$ means *undefined*). Hyperarc labels can be structured as a lattice, $(\{d, ?, \perp\}, \leq_d)$, where $d \leq_d ?$ and $\perp \leq_d d$. Let Ψ be a graph with labelled arcs and π_v^h be a path from the node v to h in Ψ. We denote by $\Lambda(\pi_v^h)$ the set of labels in the path π_v^h. This notation naturally extends to sets of hyperarcs.

Definition 3.2 *Given a dependence representation $\mathcal{D} = (\mathcal{N}, \mathcal{V}, \mathcal{E})$, a hyperarc $e \in \mathcal{E}$ is* redundant in $\mathcal{D}$ *iff $e = \langle T, h\rangle_?$ and $h \in T$, or $e = \langle T, h\rangle_l$ and $\exists T' \subseteq T,\ T \neq \emptyset,\ \exists \mathcal{E}' \subseteq \mathcal{E}$, $e \notin \mathcal{E}'$ such that $T' \rightarrow^{\mathcal{E}'}_{(\mathcal{N},\mathcal{E})} h$ and $lub_{\leq_d}(\Lambda(\mathcal{E}')) \leq_d l$.*

A hyperarc is redundant in a dependence representation iff it does not add any information to the representation itself (i.e. either it is an autodependence labelled by ?, or there exists a connection with the same target which is more concise ($T' \subseteq T$) and captures a stronger dependence relation). Notice that a connection is deterministic iff all its labels are d. In the following we will use a suitable graphical notation for dependence representations.

Example 3.1 *Let us consider the non-redundant dependence representation*

$$(v_1, ng), (v_2, ng), (v_3, ng)\ .$$

Note that the hyperarcs $\langle\{v_2, v_3\}, v_1\rangle_d$, $\langle\{v_1, v_3\}, v_2\rangle_d$ and the autodependences $\langle\{v_3\}, v_3\rangle_d$, $\langle\{v_1, v_2\}, v_2\rangle_d$, $\langle\{v_1, v_2, v_3\}, v_1\rangle_d$, $\langle\{v_1, v_2, v_3\}, v_2\rangle_d$, $\langle\{v_1, v_2, v_3\}, v_3\rangle_d$, $\langle\{v_1, v_2\}, v_1\rangle_d$, $\langle\{v_2, v_3\}, v_2\rangle_d$, $\langle\{v_2, v_3\}, v_3\rangle_d$, $\langle\{v_2\}, v_2\rangle_?$ are redundant if added to the previous dependence representation. As an example, $\langle\{v_3\}, v_3\rangle_d$ is redundant since v_3 can be reached by the node itself without considering that autodependence. Of course, all the previous dependences are redundant even if they are labelled with ?.

3.1 Abstract Atoms

Definition 3.3 *An* abstract atom *is a pair of the form $(p, \mathcal{D}_p)$ (usually denoted as $p(\mathcal{D}_p)$) where p is an n-ary predicate symbol in $Pred$ and $\mathcal{D}_p$ is a non-redundant dependence representation such that $Dom(\mathcal{D}_p) = \{1, ..., n\}$ (the nodes represent the argument positions in the atom).*

Definition 3.4 *The* abstract base, *denoted as $B^\sharp_P$, is the set of non-redundant abstract atoms $p(\mathcal{D}_p)$, for each $p \in Pred$.*

Definition 3.5 *An α-interpretation is any subset of the abstract base having at most one occurrence for each predicate symbol. We will denote such a set as $\Xi^\sharp$ ($\Xi^\sharp \subseteq 2^{B^\sharp_P}$).*

To make the analysis more concise we will return non-redundant dependences and we will approximate success patterns with their least upper bound, allowing in each α-interpretation at most one abstract atom per predicate symbol [1, 8].

Definition 3.6 *Let $p(\mathcal{D}_p)$, $p(\mathcal{D}'_p)$ be two abstract atoms, where $\mathcal{D}_p = (\{1..n\}, \mathcal{V}_p, \mathcal{E}_p)$, $\mathcal{D}'_p = (\{1..n\}, \mathcal{V}'_p, \mathcal{E}'_p)$, $\mathcal{V}_p = \{(1, t^\sharp_1), ..., (n, t^\sharp_n)\}$ and $\mathcal{V}'_p = \{(1, t'^\sharp_1), ..., (n, t'^\sharp_n)\}$.*

$$p(\mathcal{D}_p) \sqsubseteq p(\mathcal{D}'_p) \ \textit{iff}$$

- $\forall i = 1, ..., n :\ t^\sharp_i \leq_g t'^\sharp_i$

- $\forall \langle T', h\rangle_{l'} \in \mathcal{E}'_p$ *either* $t'^{\sharp}_h = ng$, $t^{\sharp}_h = g$ *and* $\forall e \in \mathcal{E}_p$: $Head(e) \neq h$ *or* $\langle T', h\rangle_{l'}$ *is redundant in* $(\{1..n\}, \mathcal{V}_p, \mathcal{E}_p \cup \{\langle T', h\rangle_{l'}\})$.

Proposition 3.1 $\sqsubseteq$ *is a partial ordering.*

Let $I_1^\sharp, I_2^\sharp \in \Xi^\sharp$, $I_1^\sharp \leq_\alpha I_2^\sharp$ iff $\forall p(\mathcal{D}_p) \in I_1^\sharp$, $\exists p(\mathcal{D}'_p) \in I_2^\sharp$: $p(\mathcal{D}_p) \sqsubseteq p(\mathcal{D}'_p)$.

Example 3.2

$\{\ q(ng, ng, ng)\} \leq_\alpha \{q(ng, ng, ng)\}$, $\{\ p(ng, g, ng, ng)\ \} \leq_\alpha \{\ p(ng, ng, ng, ng)\ \}$,

$\{\ p(ng, ng, ng, ng)\} \leq_\alpha \{\ p(ng, ng, ng, ng)\ \}$.

Since each α-interpretation includes at most one atom for each predicate symbol, it is easy to prove that $\leq_\alpha$ is a partial ordering and whenever $I_1^\sharp \subseteq I_2^\sharp$, then $I_1^\sharp \leq_\alpha I_2^\sharp$. Moreover, since $Pred$ is finite and for each n-ary predicate symbol $p \in Pred$ there always exists a finite set of abstract atoms in $B_P^\sharp$ defined on p, it is easy to prove that $(\Xi^\sharp, \leq_\alpha)$ is a finite lattice [1].

3.2 Abstract Substitutions

To develop an abstract interpretation based upon a declarative semantics, we also have to reconsider the substitution notion within the abstract framework [1].

Given a finite set of variables V, we denote by $Subst_V$ the set of substitutions ϑ defined on $\bar{V}$ for each $\bar{V} \subseteq V$. Let $\Phi_1, \Phi_2 \in 2^{Subst_V}$, $\Phi_1 \leq_S \Phi_2$ iff
$(\forall \vartheta \in \Phi_1, \exists \vartheta' \in \Phi_2 \mid \vartheta \leq \vartheta') \wedge (\forall \vartheta \in \Phi_2, \exists \vartheta' \in \Phi_1 \mid \vartheta \leq \vartheta' \ \Rightarrow\ \Phi_1 \subseteq \Phi_2)$.

Definition 3.7 *An* abstract substitution $\vartheta^\sharp$ *is a non-redundant dependence representation such that* $Dom(\vartheta^\sharp) = V$ *(the nodes are the symbols of variables in* V*).*

Thus, an abstract substitution $\vartheta^\sharp$ can be represented as a hypergraph of bindings of the form $(X/t^\sharp)$ where $X \in V$ and $t^\sharp \in U_P^\sharp$.

Given a finite set V_n of variable symbols $\{x_1, ..., x_n\}$ such that $x_i \neq x_j$ for each $i \neq j$: $Subst^\sharp_{V_n}$ denotes the set of abstract substitutions defined on the set of variables V_k, for each $V_k \subseteq V_n$. We can deal with a finite set of variables because of the bottom-up approach. In fact, in this kind of analysis the meaningful substitutions are only those referring the finite amount of variables occurring in program clauses. Let $\Sigma^\sharp_{V_n} \subseteq 2^{Subst^\sharp_{V_n}}$ such that $\forall \Phi^\sharp \in \Sigma_{V_n}$, $\Phi^\sharp$ contains at most one abstract substitution $\vartheta^\sharp$ such that $Dom(\vartheta^\sharp) = V_k$, for each $V_k \subseteq V_n$. Note that any element in $\Sigma^\sharp_V$ can be handled as an α-interpretation. As a matter of fact, abstract substitutions can be considered as abstract atoms having a different predicate symbol of arity k for each $V_k \subseteq V_n$. Thus, since the preorder $\sqsubseteq$ can be defined between any homogeneous dependence representations, independently from the domain, an ordering relation $\leq_\Sigma$ can be defined by extending the $\leq_\alpha$ on the domain of sets of abstract substitutions. It is easy to prove that [1] given a finite set of variables V_n, $(\Sigma^\sharp_{V_n}, \leq_\Sigma)$ is a finite lattice.

3.3 Galois Connections

Let $I \in 2^{B_P}$ and $p \in Pred$. In the following we denote by $I \downarrow p$ the set of atoms in I having the same predicate symbol p.

Let $I \downarrow p = \{p(t_{1_1}, ..., t_{1_n}), ..., p(t_{m_1}, ..., t_{m_n})\}$. We define an abstraction map $abs : 2^{B_P} \to \Xi^\sharp$ such that $abs(I) = \bigcup_{p \in Pred} abs(I \downarrow p)$ where $abs(I \downarrow p) = \emptyset$ if $I \downarrow p = \emptyset$ and $abs(I \downarrow p) = p(\mathcal{D}_p)$ such that $\mathcal{D}_p = (\{1..n\}, \mathcal{V}_p, \mathcal{E}_p)$ otherwise; where

- $\mathcal{V}_p = \{(i, t_i^\sharp) \mid i = 1, ..., n\}$ where $t_i^\sharp = \begin{cases} g & \text{iff } \forall j = 1, ..., m,\ Var(t_{j_i}) = \emptyset, \\ ng & \text{otherwise,} \end{cases}$

- Let $T = \{i_1, ..., i_k\}$; $\langle T, h\rangle_l \in \mathcal{E}_p$ iff $\langle T, h\rangle_l$ is non-redundant in $\mathcal{D}_p$ and
  - $\forall j = 1, ..., m,\ Var(t_{j_h}) \subseteq \bigcup_{i \in T} Var(t_{j_i})$,
  - $l = \begin{cases} d & \text{iff } \forall\ r, f \in \{1, ..., m\},\ r \neq f\ , mgu(t_{r_{i_1}}, ..., t_{r_{i_k}}, t_{f_{i_1}}, ..., t_{f_{i_k}}) = fail, \\ ? & \text{otherwise.} \end{cases}$

Example 3.3 *Let $I = \{q(a,b), q(b,b), q(c,d)\}$, then $abs(I) = \{\ q(\ g\ ,\ g\ )\}$.*

In this case, the groundness of the first argument of q uniquely determines the groundness of itself, moreover it uniquely determines the groundness of the second one. Since both the abstract arguments are ground, this information is relevant only because of the deterministic label (d) associated with the hyperarc. The (non-redundant) dependence from the second to the first argument is irrelevant and can be discarded by the analysis.

Autodependences allow to detect the source of determinism in an atom. As a matter of fact, a deterministic autodependence represents a set of arguments containing at least a deterministic term (i.e. a single choice term). The framework is designed in order to handle general (possibly auto) dependences. However, in order to let the paper more readable, in the following we will often omit any (possibly non-redundant) auto dependence in abstract objects, thus showing only the inter-argument dependences, which are more relevant for the analysis point of view.

Example 3.4 *Let $I = \{p(X, X, d), p(a, b, c)\}$, $abs(I) = \{\ p(ng, ng, g)\ \}$.*

The non-redundant autodependences $\langle\{3\}, 3\rangle_d$, $\langle\{1,2\}, 1\rangle_d$ and $\langle\{1,2\}, 2\rangle_d$ (which are not shown) show that the first two arguments and the third one deterministically determines their own groundness (the sets of argument positions $\{1,2\}$ and $\{3\}$ are the two sources of the determinism for the predicate p in I). As a matter of fact, for any concrete atom $p(t_1, t_2, t_3)$ having either the first two arguments or the third one ground, there exists at most one possiblity of unifying it with atoms in I.

Let $I = \{p(f(X,Y), Z, g(X,Y,Z)), p(a,b,c)\}$, the dependence representation associated with $abs(I)$ is shown in example 3.1, where $v_1 = 1$, $v_2 = 2$ and $v_3 = 3$.

Analogously we can define a monotonic abstraction map $abs_\Sigma : 2^{Subst_V} \to \Sigma_V^\sharp$. There are no concretization maps $conc$ such that $(abs, conc)$ is a Galois connection between interpretations and α-interpretations. Thus, in order to define a Galois connection between the concrete and the abstract domain of computation, we have to consider sets of interpretations as single objects. Let us consider the following partial ordering relations defined on sets of interpretations and α-interpretations respectively:

- $\forall D_1, D_2 \in 2^{(2^{B_P})},\ D_1 \preceq D_2$ iff $(\forall I_1 \in D_1\ \exists I_2 \in D_2 \mid I_1 \subseteq I_2)\ \wedge$ $(\forall I_2 \in D_2\ \exists I_1 \in D_1 \mid abs(I_2) \leq_\alpha abs(I_1) \Rightarrow D_1 \subseteq D_2)$,
- $\forall D_1^\sharp, D_2^\sharp \in 2^{\Xi^\sharp},\ D_1^\sharp \preceq_\alpha D_2^\sharp$ iff $(\forall I_1^\sharp \in D_1^\sharp,\ \exists I_2^\sharp \in D_2^\sharp \mid I_1^\sharp \leq_\alpha I_2^\sharp)\ \wedge\ (\forall I_2^\sharp \in D_2^\sharp,\ \exists I_1^\sharp \in D_1^\sharp \mid I_2^\sharp \leq_\alpha I_1^\sharp \Rightarrow D_1^\sharp \subseteq D_2^\sharp)$.

It is easy to prove that $(2^{(2^{B_P})}, \preceq)$ and $(2^{\Xi^\sharp}, \preceq_\alpha)$ are complete lattices.

Proposition 3.2 *The pair of functions (α, γ) such that*

- $\alpha : 2^{(2^{B_P})} \rightarrow 2^{\Xi^\sharp}$ *where* $\alpha(D) = \{abs(I) \mid I \in D\}$,
- $\gamma : 2^{\Xi^\sharp} \rightarrow 2^{(2^{B_P})}$ *where* $\gamma(D^\sharp) = \{I \mid \exists I^\sharp \in D^\sharp\ \wedge\ abs(I) = I^\sharp\}$,

holds the conditions on the Galois insertion [3] between $(2^{(2^{B_P})}, \preceq)$ *and* $(2^{\Xi^\sharp}, \preceq_\alpha)$.

3.4 Abstract Unification (α-mgu)

The *abstract most general unifier* α-$mgu : B_P^n \times B_P^{\sharp n} \rightarrow Subst_V^\sharp \cup \{fail\}$, takes two n-tuples of concrete and abstract atoms respectively and returns an abstract substitution which is defined on the set of variables of the concrete atoms. It spreads ground information on the variables of the concrete atoms and it generates a dependence between a set of variables T and a variable x iff there exists a connection, in the abstract atom, between the nodes corresponding with the variables in T and the node corresponding with a term having x as variable. The resulting dependence is deterministic if such a connection contains only deterministic labels. Furthermore, it is enough to have a deterministic connection to return a deterministic dependence.

Definition 3.8 *Let $\mathcal{E}'$ be a connection between a set of nodes $\mathcal{N}'$ and a node $n \in \mathcal{N}$ in a dependence representation $\mathcal{D} = (\mathcal{N}, \mathcal{V}, \mathcal{E})$. $\mathcal{E}'$ is* relevant *iff $lub_{\leq_d}(\Lambda(\mathcal{E}')) \neq \bot$.*

The notion of minimality naturally extends to relevant connections (i.e. a *minimal relevant connection* is any connection $\mathcal{E}'$ such that $\forall e \in \mathcal{E}',\ \mathcal{E}'/\{e\}$ is not a relevant connection). Let us consider the concrete and the abstract atoms, $p(t_1, ..., t_n)$ and $p(\mathcal{D}_p)$ respectively, where $\mathcal{D}_p = (\{1..n\}, \mathcal{V}_p, \mathcal{E}_p)$ and let $R \subseteq \{1, ..., n\}$ be the set of indexes such that $\forall i \in R,\ Var(t_i) = \emptyset$. We define a dependence representation associated with the unification process: $\mathcal{D}_{\alpha-mgu} = (\mathcal{N}_{\alpha-mgu}, \mathcal{V}_{\alpha-mgu}, \mathcal{E}_{\alpha-mgu})$ where

- $\mathcal{N}_{\alpha-mgu} = \{1, ..., n\} \cup Var(p(t_1, ..., t_n)) \cup \{\otimes_i \mid i \in R\}$,
- $\mathcal{V}_{\alpha-mgu} = \mathcal{V}_p \cup \{(\otimes_i, g) \mid \exists i \in \{1..n\} : Var(t_i) = \emptyset\} \cup \{(x, ng) \mid \exists x \in Var(t_1..t_n)\}$;
- $\mathcal{E}_{\alpha-mgu} = \mathcal{E}_p\ \cup\ \{\langle\{\otimes_i\}, i\rangle_\bot, \langle\{i\}, \otimes_i\rangle_\bot \mid \exists i \in \{1..n\} : Var(t_i) = \emptyset\}\ \cup$ $\{\langle\{x_1, ..., x_k\}, i\rangle_\bot, \langle\{i\}, x_1\rangle_\bot, ..., \langle\{i\}, x_k\rangle_\bot \mid \exists i \in \{1..n\} : Var(t_i) = \{x_1, ..., x_k\}\}$.

The abstract unification process works on this dependence representation and consists of the following two steps:

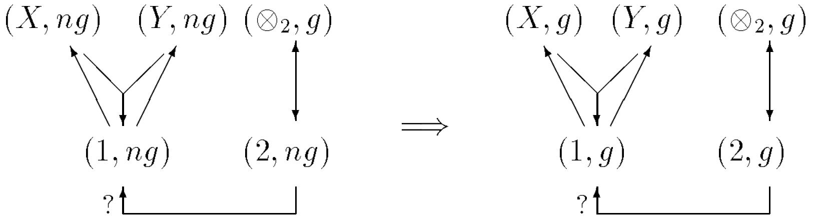
1. *Ground information propagation*
   It is an iterative process on the hypergraph $\mathcal{G}_{\alpha-mgu} = (\mathcal{N}_{\alpha-mgu}, \mathcal{E}_{\alpha-mgu})$ which changes the labels in $\mathcal{V}_{\alpha-mgu}$. The process replaces any labelled node $(v, ng) \in \mathcal{V}_{\alpha-mgu}$ with (v, g) iff there exists a hyperarc $e \in \mathcal{E}_{\alpha-mgu}$ such that $Head(e) = v$ and $\forall v_i \in Tail(e): \ (v_i, g) \in \mathcal{V}_{\alpha-mgu}$ [1]. The process halts when no more labels can be changed.

2. *α-substitution definition*
   This process considers the transformed hypergraph and returns an abstract substitution $\vartheta^\sharp = (Var(t_1, ..., t_n), \mathcal{V}_\vartheta, \mathcal{E}_\vartheta)$ such that
   $\mathcal{V}_\vartheta = \{(x, t^\sharp) \in \mathcal{V}_{\alpha-mgu} \mid x \in Var(t_1, ..., t_n)\}$,
   $\forall x_k \in Var(t_1, ..., t_n)$, $\langle T, x_k\rangle_l \in \mathcal{E}_\vartheta$ iff $\langle T, x_k\rangle_l$ is non-redundant in $\vartheta^\sharp$ and if $\bar{\mathcal{E}}$ is the set of minimal relevant connections from T to x_k, then $\bar{\mathcal{E}} \neq \emptyset$ and $l = glb_{\leq_d}\left\{ \ lub_{\leq_d}(\Lambda(\mathcal{E}')) \,\middle|\, \mathcal{E}' \in \bar{\mathcal{E}} \ \right\}$.

The condition $lub_{\leq_d}(\Lambda(\pi)) \neq \bot$ avoids the introduction of irrelevant hyperarcs, labelled by $\bot$. As a matter of fact, we are only interested in those connections $\mathcal{E}'$ containing at least a hyperarc in $\mathcal{E}_p$. Notice that $\mathcal{D}_{\alpha-mgu}$ can be redundant.

Example 3.5 *Let* $\mathcal{D}_{\alpha-mgu} = \begin{array}{c}(X, ng)\\ \updownarrow\bot \\ (1, ng)\end{array}$ *, and* $\mathcal{E}'_1 = \{\langle\{X\}, 1\rangle_\bot, \langle\{1\}, X\rangle_\bot, \langle\{1\}, 1\rangle_?\}$, $\mathcal{E}'_2 = \{\langle\{X\}, 1\rangle_\bot, \langle\{1\}, X\rangle_\bot\}$.
Note that $\forall i = 1, 2: \ \{X\} \rightarrow^{\mathcal{E}'_i}_{\mathcal{D}_{\alpha-mgu}} X$*, but since* $\mathcal{E}'_2$ *contains only undefined* ($\bot$) *hyperarcs, only* $\mathcal{E}'_1$ *is "relevant" in the unification process.*

Example 3.6 *The ground information propagation applied to the dependence representation associated with the abstract unification of* $p(f(X, Y), a)$ *and* $p(ng, ng)$ *is* ($\bot$ *labels are omitted):*

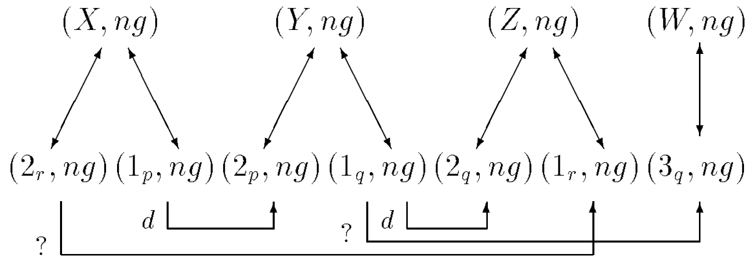


It is easy to extend the previous abstract unification algorithm to any n-tuple ($n \geq 1$) of atoms, as shown in the following example.

Example 3.7 *Let us consider the abstract unification of the concrete and abstract atoms*

$$p(X, Y), q(Y, Z, W), r(Z, X) \ \textit{and} \ p(ng, ng) \ , \ q(ng, ng, ng) \ , \ r(ng, ng) \ .$$

Since there exist shared variables between the concrete atoms $p(X, Y)$, $q(Y, Z, W)$, $r(Z, X)$, *the unification hypergraph contains multiple connections between the corresponding variables and the abstract terms (the predicate symbol is subscripted in any corresponding argument position). The corresponding dependence representation is*



The resulting abstract substitution is: $(Z/ng, Y/ng, X/ng, W/ng)$.

Proposition 3.3 (α-*mgu* **monotonicity)**
Let A be an atom. Given a pair of abstract atoms $A^\sharp$, $A'^\sharp$ such that $\{A^\sharp\} \leq_\alpha \{A'^\sharp\}$, α-$mgu(A, A^\sharp) \leq_\Sigma \alpha$-$mgu(A, A'^\sharp)$.

3.5 Substitution Application (α-*apply*)

In order to develop a deterministic ground dependence analysis we have to define the application of abstract substitutions (returned by the abstract unification process) to concrete atoms, i.e. α-$apply : 2^{(B_P \times Subst^\sharp_V)} \rightarrow \Xi^\sharp$. Given a set of pairs of concrete atoms (having the same predicate symbol) and abstract substitutions, it returns an abstract atom which collects the deterministic ground information belonging to this set.

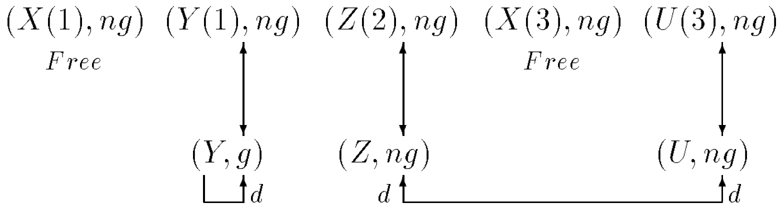
Let $S \in 2^{(B_P \times Subst^\sharp_V)}$ such that $S \downarrow p = \{(p(t_{1_1}, ..., t_{1_n}), \vartheta^\sharp_1), ..., (p(t_{m_1}, ..., t_{m_n}), \vartheta^\sharp_m)\}$, and let $\mathcal{N}_{i_j} = \{x(j) \mid x \in Var(t_{i_j})\}$ be the set of variables in t_{i_j}, indexed according to the corresponding argument position.

For each pair $(p(t_{i_1}, ..., t_{i_n}), \vartheta^\sharp_i)$ where $\vartheta^\sharp_i = (\mathcal{N}_{\vartheta_i}, \mathcal{V}_{\vartheta_i}, \mathcal{E}_{\vartheta_i})$, we generate a dependence representation $\mathcal{D}_{\alpha-apply_i} = (\mathcal{N}_{\alpha-apply_i}, \mathcal{V}_{\alpha-apply_i}, \mathcal{E}_{\alpha-apply_i})$ such that:

- $\mathcal{N}_{\alpha-apply_i} = \mathcal{N}_{\vartheta_i} \cup (\bigcup_{j=1}^{n} \mathcal{N}_{i_j})$;
- $\mathcal{V}_{\alpha-apply_i} = \mathcal{V}_{\vartheta_i} \cup \{(x(j), ng) \mid \exists j = 1..n : x(j) \in \mathcal{N}_{i_j}\}$;
- $\mathcal{E}_{\alpha-apply_i} = \mathcal{E}_{\vartheta_i} \cup \{\langle\{x(j)\}, x\rangle_\perp, \langle\{x\}, x(j)\rangle_\perp \mid \exists (x, t^\sharp) \in \mathcal{V}_{\vartheta_i}, \exists j = 1..n : x(j) \in \mathcal{N}_{i_j}\}$ (if $\exists (x, t^\sharp) \in \mathcal{V}_{\vartheta_i}$, $\exists j = 1..n :$ $x(j) \in \mathcal{N}_{i_j}$, $x(j)$ is a *linked variable*).

Example 3.8 *The dependence representation associated with the pair*

$(p(f(X, Y), Z, f(X, U)),\ (Z/ng, U/ng, Y/g)\ )$ *is:*



Let us consider a ground information propagation on $\mathcal{G}_{\alpha-apply_i} = (\mathcal{N}_{\alpha-apply_i}, \mathcal{E}_{\alpha-apply_i})$, $i = 1, ..., m$, which is analogous to the α-mgu case. We define

$$\alpha\text{-}apply(S) = \bigcup_{p \in Pred} \alpha\text{-}apply(S \downarrow p)$$

where $\alpha\text{-}apply(S \downarrow p) = \begin{cases} \emptyset & \text{if } S \downarrow p = \emptyset \\ p(\mathcal{D}_p) & \text{otherwise} \end{cases}$, such that $\mathcal{D}_p = (\{1..n\}, \mathcal{V}_p, \mathcal{E}_p)$ and

- $$\forall (j, t_j^\sharp) \in \mathcal{V}_p : \ t_j^\sharp = \begin{cases} g & \text{iff} \quad \forall i = 1, ..., m : \ \mathcal{N}_{i_j} = \emptyset \\ & \qquad\ \ \text{(the term was already ground)} \\ & \qquad\ \ \text{or the nodes in } \mathcal{N}_{i_j} \text{ are labelled } g, \\ ng & \text{otherwise,} \end{cases}$$

- $e = \langle \{r_1, ..., r_k\}, h \rangle_l \in \mathcal{E}_p$ iff e is non-redundant in $\mathcal{D}_p$ and for each $i = 1, ..., m$:
  - $Free(\mathcal{N}_{i_h}) \subseteq \bigcup_{u=r_1}^{r_k} Free(\mathcal{N}_{i_u})$, where $Free(\mathcal{N}_{i_j})$ is the set of variables x such that $x(j)$ is not linked with any node in the hypergraph $\mathcal{G}_{\alpha-apply_i}$;
  - for each linked node $y(h) \in \mathcal{N}_{i_h}$, there exists $T \subseteq \bigcup_{u=r_1}^{r_k} \mathcal{N}_{i_u}$ such that, if $\bar{\mathcal{E}}^T_{i_{y(h)}}$ is the set of minimal relevant connections from T to $y(h)$ in $\mathcal{G}_{\alpha-apply_i}$, then $\bar{\mathcal{E}}^T_{i_{y(h)}} \neq \emptyset$;
  - $$l = \begin{cases} d & \text{iff } \forall z, f \in \{1, ..., m\}, z \neq f : mgu(t_{z_{r_1}}, ..., t_{z_{r_k}}, t_{f_{r_1}}, ..., t_{f_{r_k}}) = fail, \\ & \quad \exists y(h) \in \mathcal{N}_{i_h}, \ \exists T \subseteq \bigcup_{u=r_1}^{r_k} \mathcal{N}_{i_u} : \ \bar{\mathcal{E}}^T_{i_{y(h)}} \neq \emptyset \ \wedge \\ & \quad d = glb_{\leq_d} \left\{ lub_{\leq_d}(\Lambda(\mathcal{E}')) \,\middle|\, \mathcal{E}' \in \bar{\mathcal{E}}^T_{i_{y(h)}}) \right\}, \\ ? & \text{otherwise.} \end{cases}$$

Example 3.9 *The following examples show the results of the application of the α-apply operator in some interesting cases:*

$S_1 = \{(p(X, Z), \ (X/ng, Z/ng) \ ), (p(f(X), K), \ (X/ng, K/ng) \ )\}, \ \alpha\text{-}apply(S_1) =$

$\{ \ p(ng, ng) \ \};$

$S_2 = \{(p(f(X, Y), Z, f(X, U)), \ (Z/ng, U/ng, Y/g) \ ), (p(0, X, X), \emptyset)\}, \ \alpha\text{-}apply(S_2) =$

$\{ \ p(ng, ng, ng) \ \}$ *(see example 3.8).*

Let I be a finite set of atoms, and $S = I \times \{\emptyset\}$. By *$\alpha$-apply* definition we have $\alpha\text{-}apply(S) = abs(I)$.

Proposition 3.4 (α-*apply* **monotonicity)**
Given two sets of pairs of concrete atoms (having the same predicate symbol) and abstract substitutions $\{(A_1, \vartheta_1^\sharp), ..., (A_k, \vartheta_k^\sharp)\}$ and $\{(A_1, \vartheta_1^{'\sharp}), ..., (A_k, \vartheta_k^{'\sharp})\}$, such that $\forall i = 1, ..., k : \ \vartheta_i^\sharp \leq_\Sigma \vartheta_i^{'\sharp}$, it follows that

$$\alpha\text{-}apply(\{(A_i, \vartheta_i^\sharp) \mid i = 1, ..., k\}) \leq_\alpha \alpha\text{-}apply(\{(A_i, \vartheta_i^{'\sharp}) \mid i = 1, ..., k\}).$$

3.6 Abstract Interpretation

The approximation of the concrete fixpoint semantics is given in terms of the fixpoint of a finitely converging monotonic operator defined in terms of α-*mgu* and α-*apply*.

Definition 3.9 *Given a logic program P, let $I^\sharp$ be an interpretation. The abstract immediate consequence operator associated with the program P, $T_P^\sharp : \Xi^\sharp \to \Xi^\sharp$, is defined as follows*

$$T_P^\sharp(I^\sharp) = \alpha\text{-}apply(\left\{ (p(\bar{t}), \vartheta^\sharp) \;\middle|\; \begin{array}{l} \exists p(\bar{t}) : -B_1, ..., B_n \in P,\ \exists B_1'^\sharp, ..., B_n'^\sharp \in I^\sharp, \\ \vartheta^\sharp = \alpha - mgu(B_1, ..., B_n, B_1'^\sharp, ..., B_n'^\sharp), \\ \vartheta^\sharp \neq fail \end{array} \right\}).$$

The correctness of $T_P^\sharp$ ($abs(T_P(I)) \leq_\alpha T_P^\sharp(abs(I))$) follows by the correctness of α-*mgu* and α-*apply*. Let $lfp(f)$ denote the least fixpoint of a given function f. Since the lattice $(\Xi^\sharp, \leq_\alpha)$ is finite, there exists a finite positive number k such that $\mathcal{F}^\sharp(P) = lfp(T_P^\sharp) = T_P^\sharp \uparrow k$.

Let us consider the concrete and the abstract operators

- $\mathcal{T}_P : 2^{(2^{B_P})} \to 2^{(2^{B_P})}$, such that $\mathcal{T}_P(D) = \{T_P(I) \mid I \in D\}$,
- $\mathcal{T}_P^\sharp : 2^{\Xi^\sharp} \to 2^{\Xi^\sharp}$ such that $\mathcal{T}_P^\sharp(D^\sharp) = \{T_P^\sharp(I^\sharp) \mid I^\sharp \in D^\sharp\}$.

It is esay to prove that for each logic program P, $\{\mathcal{F}(P)\}$ and $\{\mathcal{F}^\sharp(P)\}$ are fixpoints of $\mathcal{T}_P$ and $\mathcal{T}_P^\sharp$ respectively ($\mathcal{T}_P \uparrow \omega = \{\mathcal{F}(P)\}$ and $\mathcal{T}_P^\sharp \uparrow k = \{\mathcal{F}^\sharp(P)\}$). Thus, by $T_P^\sharp$ correctness: $\alpha(\mathcal{T}_P \uparrow \omega) \preceq_\alpha \mathcal{T}_P^\sharp \uparrow k$ (or equivalently [3] $\mathcal{T}_P \uparrow \omega \preceq \gamma(\mathcal{T}_P^\sharp \uparrow k)$).

4 Applications

In the following example [14] we show the overall analysis by describing the composition of the operators defined in Section 3. Consider the predicate r defined as follows

$$r(g(X), g(X)) : -.$$

$$r(f(X), g(Y)) : -r(Y, X).$$

Let us examine the behaviour of our analysis:

$$T_r^{\sharp^1}(\emptyset) = \alpha\text{-}apply(\{(r(g(X), g(X)), \emptyset)\}) = \{\ r(ng, ng)\}.$$

In the first step, the abstract unification returns the pair $(r(g(X), g(X)), \emptyset)$ only. A ground dependence is detected between the first and the second argument and on the other way round by α-*apply*. Since the argument of α-*apply* is a set of pairs including only one element, the resulting dependences are labelled as deterministic.

$$T_r^{\sharp^2}(\emptyset) = \alpha\text{-}apply(\{(r(g(X), g(X)), \emptyset), (r(f(X), g(Y)), (X/ng, Y/ng)\ )\} = \{\ r(ng, ng)\}.$$

In the second step, α-*apply* considers two pairs: the one returned by the previous step and the one produced by the abstract unification of the abstract atom returned by the previous step with the concrete atom in the body of the non-unit clause of r. Since the second arguments of the concrete atoms in the pairs unify, this step returns a single deterministic dependence from the first argument to the second one.

$$T_r^{\sharp^3}(\emptyset) = \alpha\text{-}apply(\{(r(g(X),g(X)),\emptyset),(r(f(X),g(Y)),\ (X/ng, Y/ng)\ )\} = \{\ r(ng, ng)\}.$$

The third step (fixpoint) detects that the dependence between the first and the second argument is a non-deterministic one. As in the previous steps, the first two arguments do not unify, but, since no deterministic dependence exists between the abstract terms associated with the variables X and Y, no deterministic dependence is returned. Let us consider the program

$sumlist([], 0) : -.$

$sumlist([X|Xs], S) : -sumlist(Xs, Ss), plus(X, Ss, S).$

$select(X, [X|Xs], Xs) : -.$

$select(X, [Y|Ys], [Y|Zs]) : -select(X, Ys, Zs).$

$check(X, Y) : -select(Y, X, R), sumlist(X, Y).$

The predicate $sumlist(X, Y)$ is such that Y is the sum of the elements of the list X. The predicate *plus* is assumed to be a predefined one. The dependences of *plus* relevant for the analysis are a deterministic one from the first and the second argument to the third one and a non-deterministic one from the third argument to the first and the second, respectively. The list X includes relative numbers. The predicate $select(X, L_1, L_2)$ is such that L_2 is the list L_1 where exactly one occurrence of X has been removed. The predicate $check(X, Y)$ is such that Y is an element E of the list X and E is equal to the sum of the elements of X. The dependences detected by the analysis of *sumlist* and of *select* may be easily deduced from the declarative meaning of the predicates (only some dependences of *sumlist* and *select* are shown):

$$sumlist(ng, ng) \qquad select(ng, ng, ng).$$

The dependences of *sumlist* and *select* are propagated, through abstract unification, to the body of *check*. In the hypergraph built by abstract unification, the variable Y is the target of two paths. Both of them have the variable X as source node, but only one is labelled by d. In this case, a deterministic dependence between the variables X and Y is returned. Therefore, there exists a deterministic dependence between the first and the second argument of *check*. As a matter of fact, since the sum of the elements in the list is uniquely determined, all the elements selected from the list are equivalent, hence the dependence may be considered as a deterministic one. The analysis infers deterministic dependences among arguments rather than determinacy of whole predicates. Let us consider the following modified version of predicate check:

$check(X, Y, R) : -select(Y, X, R), sumlist(X, Y).$

In this case also a non-deterministic dependence is detected between the first argument and the third one. As a matter of fact, if a list includes several elements equal to its sum, distinct lists are returned even by removing the same element.

$$check(ng, ng)\ , \quad check(ng, ng, ng)\ .$$

The two former examples show that our analysis can return deterministic dependences not detected by previous analyses. In particular, the last example shows the importance of defining deterministic dependences among arguments rather than for the whole predicate.

Let us finally consider the classical example:

$$append([\,], X, X) : -.$$

$$append([X|Xs], Y, [X|Ys]) : -append(Xs, Y, Ys).$$

The analysis detects the following dependences: $append(ng, ng, ng)$ (see Example 3.8 and 3.9).

5 Conclusions

Our analysis of determinate computations extends previous proposals in several directions. The ground dependence notion, introduced in the database framework [11], has been extended in order to handle recursive clauses. Hence, the analysis returns significative results not only in the case of database programming, but also in other cases, such as programs manipulating recursive data structures. We recall that determinacy is inferred with respect to arguments rather than to whole predicates. This means that a ground instance of a set of arguments may uniquely determine another set of arguments of a predicate, while producing multiple bindings for other arguments. With respect to [14], our proposal can detect both ground and ground deterministic dependences in a single step, and requires no knowledge of the types of the predicates nor of the activating modes of a predicate. Our abstract interpretation can be easily integrated with further analyses, such as depth-k [13], type and general functional ones. By pairing our analysis with a term structure one (e.g. depth-k), dependences among subterms may be detected. Furthermore, it is possible to relax the ground dependences condition in order to obtain a more general analysis of functional dependences.

References

[1] 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, 1991. To appear in *ACM Transactions on Programming Languages and Systems*.

[2] C. Berge. *Graphs and Hypergraphs*. North-Holland, 1973.

[3] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. Fourth ACM Symp. Principles of Programming Languages*, pages 238–252, 1977.

[4] S. Debray and D.S. Warren. Functional Computations in Logic Programs. *ACM Transactions on Programming Languages and Systems*, 11-3:451–481, 1989.

[5] B. Demoen, P. VanRoy, and Y.D. Willems. Improving the Execution Speed of Compiled Prolog with Modes, Clause Selection and Determinism. In H. Ehrig, R. Kowalski, G. Levi, and U. Montanari, editors, *Proc. TAPSOFT 1987*, volume 250 of *Lecture Notes in Computer Science*, pages 111–125. Springer-Verlag, Berlin, 1987.

[6] P. Deransart and J. Maluszynski. Relating Logic Programs and Attribute Grammars. *Journal of Logic Programming*, 2:119–156, 1985.

[7] 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.

[8] R. Giacobazzi and L. Ricci. Pipeline Optimizations in AND-Parallelism by Abstract Interpretation. In D. H. D. Warren and P. Szeredi, editors, *Proc. Seventh Int'l Conf. on Logic Programming*, pages 291–305. The MIT Press, Cambridge, Mass., 1990.

[9] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1987. Second edition.

[10] M.J. Maher and R. Ramakrishnan. Dèjà Vu in Fixpoints of Logic Programs. In E. Lusk and R. Overbeck, editors, *Proc. North American Conf. on Logic Programming'89*, pages 963–980. The MIT Press, Cambridge, Mass., 1989.

[11] A.O. Mendelzon. Functional Dependencies in Logic Programs. In *Proc. of the Eleventh International Conference on Very Large Data Bases*, pages 324–330, 1985.

[12] L. Ricci. *Compilation of Logic Programs for Massively Parallel Systems.* PhD thesis, Universitá di Pisa, Feb. 1990. T.D. 3-90.

[13] T. Sato and H. Tamaki. Enumeration of Success Patterns in Logic Programs. *Theoretical Computer Science*, 34:227–240, 1984.

[14] J. Zobel and P. Dart. On Logic Programs, Functional Dependencies, and Types. Technical report, University of Melbourne, 1990.