As will turn out, any $\forall x\, p$-formula is decomposable as a finite disjunction $\bigvee_{t\in\mathcal{T}_p} \Phi_t$, where each Φ_t (involving the same free variables as p —namely $x_1, \ldots, x_n$—, and the bound variable x) is satisfiable, whereas no two Φ_t's are satisfiable together. In a sense, this is an irredundant decomposition of $\forall x\ p$, and hence the unsatisfiability of $\forall x\, p$ can be established by simply checking whether or not $\mathcal{T}_p$ is empty.

Such a decomposition can be carried out even under weak axiomatizations insufficient to ensure the completeness of $\exists x_1 \cdots \exists x_n\, \forall x\, p$-sentences. Three such 'weak' theories, named $\mathcal{S}, \mathcal{S}_E$, and $\mathcal{S}_R$, will be introduced in this paper.

An usage of the proposed decomposition of $\forall x p$ is the following. Suppose one is assigned the task of proving many implications of the form $\forall x\, p \rightarrow \forall x\, p_i$, with common antecedent $\forall x\, p$, where each p_i is unquantified and involves the same free variables as p —namely x, $x_1, \ldots, x_n$—. As will turn out, 'evaluating' each $\forall x p_i$ in Φ_t, for all t in $\mathcal{T}_p$, is possible thanks to the particular structure of the disjuncts Φ_t. One is authorized to conclude that $\forall x\, p$ yields $\forall x\, p_i$ if and only if $\forall x\, p_i$ evaluates to *TRUE* in every disjunct Φ_t of $\forall x\, p$.

An issue left open by this paper is how to optimize the satisfiability test which simply consists in determining the presence of a disjunct in the decomposition of $\forall x\, p$, in analogy with the analysis that has already been carried out (cf. [COP89]) for the fragment of set theory underlying multilevel syllogistic.

It is conjectured that the satisfiability problem for the $\forall x\, p$ set theoretic formulae is NP-complete, like the analogous problem for multilevel syllogistic and some of its extensions (refer again to [COP89]).

For the time being, only a very coarse estimate of the complexity of this problem can be inferred from Corollary 5 in Section 5.

2 Axioms

Let us begin by setting up in first-order predicate calculus with equality a small theory of sets, $\mathcal{S}$, whose axioms state that an empty set exists and that a set is always obtained by adding or removing an element y to (from) a set x.

$$\begin{array}{lll} \exists z & \forall v & v \notin z, \\ \exists w & \forall v & (v \in w \ \leftrightarrow\ v \in x \ \vee\ v = y), \\ \exists \ell & \forall v & (v \in \ell \ \leftrightarrow\ v \in x \ \&\ v \neq y). \end{array}$$

Despite its rudimentariness, this theory yields non-obvious theorems, such as

$$\left(\&_{i=1}^{n-1}\&_{j=i+1}^{n} x_i \neq x_j\right) \rightarrow \exists x \left(\&_{i=1}^{k}(x \neq x_i \& x_i \in x) \,\&\, \&_{j=k+1}^{n}(x \neq x_j \& x_j \notin x)\right),$$

where $k \leq n$. In order to see this, we first skolemize the axioms, which become

$$v \notin \emptyset,\ v \in x \,\texttt{with}\, y \leftrightarrow (v \in x \vee v = y),\ v \in x \,\texttt{less}\, y \leftrightarrow (v \in x \,\&\, v \neq y),$$

then we introduce the abbreviation

$$\{t_1, \ldots, t_h\} \;=\; \emptyset \,\texttt{with}\, t_1 \,\texttt{with}\, \cdots \,\texttt{with}\, t_h,$$

and finally proceed to construct the desired x as follows[4]:

[4]This pseudo-algorithm must be viewed as the detailed plan of how to carry out the proof of the thesis for each given pair n, k. Only its initial assignment $x{:=}\{x_1, \ldots, x_k\}$ needs to be retained when extensionality and regularity axioms —see below— are adopted.


```
x := {x_1, ..., x_k};
choose z from among ∅, {∅}, {{∅}}, ..., {^n ∅}^n
        such that z differs from x_1, ..., x_n;
for i = 1, ..., n
    if z ∉ x_i then x := x with z; else
        if ¬∀z_0 ∀z_1 ··· ∀z_{n+1} ((z_0 ∈ x_i & &^n_{j=0} &^{n+1}_{k=j+1} z_j ≠ z_k &
                &^{n+1}_{j=1} z_j = {z_{j-1}}) → V^{n+1}_{j=1} z_j ∈ x_i)
        then
            choose z_0, z_1, ..., z_{n+1} with
                    z_0 ∈ x_i & &^n_{j=0} &^{n+1}_{k=j+1} z_j ≠ z_k &
                    &^{n+1}_{j=1} (z_j = {z_{j-1}} & z_j ∉ x_i)
            choose y from among z_1, ..., z_{n+1}
                    such that y differs from x_1, ..., x_n;
            x := x with y;
        else
            comment: x_i is infinite and hence it does not coincide
            —nor will at any later step— with the value of x;
        end if;
    end if;
end for.
```

One further axiom, namely the *extensionality axiom*

(E) $\quad \forall v\,(v \in x \leftrightarrow v \in y) \rightarrow x = y,$

leads from $\mathcal{S}$ to the theory $\mathcal{S}_E$, which was studied in [PP88] under a different name, T_0.

It is usually assumed that membership forms no cycles:

$$\neg\, x_0 \in x_1 \in \cdots \in x_n \in x_0.$$

Instead of postulating the infinitely many instances of this scheme that result as n varies over $0, 1, 2, \ldots$, it is usually preferred to adopt the single axiom

(R) $\quad v \in x \rightarrow \exists m\,(m \in x \,\&\, \forall w\,(w \in x \rightarrow w \notin m)),$

called the *regularity axiom*, which entails the preceding scheme. By adding (R) to $\mathcal{S}$ and to $\mathcal{S}_E$, one obtains $\mathcal{S}_R$ and $\mathcal{S}_{ER}$ respectively.

We notice in passing a fact which is useful in the proof of a later theorem (Theorem 4 in Section 5): by just adding the assignment

$$x := x \,\texttt{with}\, \{x_1, \ldots, x_n\}$$

as the final step in a construction —for instance the one seen above— of an x satisfying $\left(\&_{i=1}^{k} x_i \in x\right)$ & $\left(\&_{j=k+1}^{n} x_j \notin x\right)$, one will achieve also $x \notin x_\ell$, $x \neq x_\ell$, for $\ell = 1, \ldots, n$.

3 A succint skolemization of the extensionality and regularity axioms

One of many ways of expressing **(E)** and **(R)** in clausal form is by introducing a Skolem function $\eta(\bullet,\bullet)$ subject to the rules:

(E′) $\quad \eta(x,x) \in x \vee x = \emptyset$,

(E″) $\quad \eta(x,y) \notin x \vee \eta(x,y) \notin y \vee x = y$,

(E‴) $\quad \eta(x,y) \in x \vee v \notin y \vee \eta(y,x) \in y$,

(R′) $\quad v \notin x \vee \eta(x,x) \in x$,

(R″) $\quad v \notin x \vee v \notin \eta(x,x)$,

where **(R′)** follows from **(E‴)** when y is instantiated as x in the latter.

This skolemization has not been carried out in the most straightforward manner, because we wanted to express our theories very shortly. Let us now justify what has been produced.

If **(E)** is postulated, then plainly $x \neq \emptyset$ implies $\exists v\, v \in x$. One can therefore introduce an η such that $\eta(x,x) \in x$ when $x \neq \emptyset$, and it will be unproblematic to require at the same time that

(E$_*$) $\quad \eta(x,y) \in x \,\&\, \eta(x,y) \notin y$ when $\exists v(v \in x \,\&\, v \notin y)$,

as the latter situation clearly presupposes $x \neq y$. Taking another consequence of **(E)** into account, namely that

$$x \subsetneqq y, \text{ i.e. } x \neq y \,\&\, \neg\exists v(v \in x \,\&\, v \notin y), \text{ implies } \exists w(w \in y \,\&\, w \notin x),$$

one can impose on η the additional constraint

(E*) $\quad \eta(x,y) = \eta(y,x)$ when $x \subsetneqq y$,

to the effect that $x \neq y$ imply $\eta(x,y) \in x \not\leftrightarrow \eta(x,y) \in y$ in all cases and that **(E′)**, **(E″)**, **(E‴)** be fulfilled.

Conversely, if **(E′)**, **(E″)** and **(E‴)** are postulated, then **(E)** can be proved. Indeed, as we will now show, $x \neq y$ implies $\exists v\,(v \in x \not\leftrightarrow v \in y)$ in each of the four possible cases:

1. $(\neg\ \exists z\ \ z \in x) \,\&\, (\neg\ \exists w\ \ w \in y)$,
2. $(\neg\ \exists w\ \ w \in x) \,\&\, (\ \ \exists v\ \ v \in y)$,
3. $(\ \ \exists v\ \ v \in x) \,\&\, (\neg\ \exists w\ \ w \in y)$,
4. $(\ \ \exists z\ \ z \in x) \,\&\, (\ \ \exists w\ \ w \in y)$.

One has in fact, respectively:

1. $x \neq y$ cannot hold, as $x = \emptyset$ and $y = \emptyset$ by **(E′)**.
2. Any $v \in y$ satisfies the desired formula $v \in x \not\leftrightarrow v \in y$.
3. Any $v \in x$ satisfies the desired formula $v \in x \not\leftrightarrow v \in y$.

4. $\eta(x,y) \notin x \vee \eta(x,y) \notin y$ follows from $x \neq y$ and **(E″)**; hence $\eta(x,y)$ is the desired v satisfying $v \in x \not\leftrightarrow v \in y$ if $\eta(x,y) \in x$. If $\eta(x,y) \notin x$ instead, then $\eta(y,x) \in y$ by **(E‴)** (applicable, because $w \in y$), and hence $\eta(y,x) \notin x$ (by **(E″)**, with roles of x and y interchanged); hence $\eta(y,x)$ is the desired v.

Finally, if **(R)** is postulated, then one can introduce an η fulfilling **(R′)** and **(R″)**. These two clauses in turn yield **(R)**; also, they are clearly compatible with **(E′)**, **(E$_*$)** and **(E*)**, and hence with **(E′)**, **(E″)**, **(E‴)**.

4 The type of an n-tuple of sets, and the formula it expresses

As explained in the introduction, our goal is to decompose $\forall x\, p$, where p is any unquantified formula involving the free variables $x, x_1, \ldots, x_n$, as a disjunction $\bigvee_{t \in \mathcal{T}_p} \Phi_t$. The Φ_t's are to satisfy a number of properties —see Section 6—; in particular, none of them should be refutable. The decomposition will be carried out differently in different theories, fewer axioms requiring more disjuncts. Thanks to the properties of logical equality, it will not be restrictive to assume that p has the form[5]

(*) $\qquad \left(\&_{i=1}^{n-1} \&_{j=i+1}^{n} x_i \neq x_j\right) \,\&\, \left((\&_{j=1}^{n} x \neq x_j) \rightarrow p_*\right)$,

where p_* does not contain equality.

DEFINITION 1 *Given n distinct sets $a_1, \ldots, a_n$, put*

$$G = \{(i,j) : i,j = 1,\ldots,n \,|\, a_i \in a_j\}.$$

Consider also the family Γ formed by all relations of the form

$$G \cup \{(0,j) : j = 1,\ldots,n \,|\, a \in a_j\} \cup \{(j,0) : j = 1,\ldots,n \,|\, a_j \in a\},$$

obtained by letting a vary over all sets different from $a_1, \ldots, a_n$. Although a varies over a universe which is presumably very vast, Γ is finite: actually, its size cannot exceed 2^{n+1^2}, because $\Gamma \subseteq \mathcal{P}(\{0,\ldots,n\}^2)$.

The family $\{G\} \cup \Gamma$ is called n-type induced by $a_1, \ldots, a_n$. □

Here we attempt to characterize an n-type independently of an inducing tuple of sets. Our first definition is very coarse, as it still awaits to be tuned —see the definition of T-filter below— to the particular theory ($\mathcal{S}$, $\mathcal{S}_E$, $\mathcal{S}_R$, $\mathcal{S}_{ER}$, or a stronger theory) one has to deal with.

[5] Technically, that we can make this restriction is due to the availability of the theorem $\exists y_N \&_{i=0}^{N-1} y_i \neq y_N$ in our theories. Thanks to it, any formula $\forall y_N q$ with matrix q involving the distinct variables $y_0, \ldots, y_N$ can be decomposed as

$$\forall y_N\, q \leftrightarrow \bigvee_{\varrho} \left(\left(\&_{i=0}^{N-1} y_i = y_{\varrho i}\right) \,\&\, \forall x\, p_\varrho\right)$$

where: ϱ ranges over the functions $\{0,\ldots,N-1\} \rightarrow \{0,\ldots,N-1\}$ such that $\varrho\varrho i = \varrho i \leq i$ for all i; p_ϱ has the form (*), with x, $\{x_1,\ldots,x_n\}$ and p_* instantiated as y_N, $\{y_{\varrho 1},\ldots,y_{\varrho N}\}$ and $p \begin{smallmatrix} y_0 & \cdots & y_{N-1} \\ y_{\varrho 0} & \cdots & y_{\varrho(N-1)} \end{smallmatrix}$ $\&$ $\&_{i=0}^{N-1} p \begin{smallmatrix} y_0 & \cdots & y_{N-1} & y_N \\ y_{\varrho 0} & \cdots & y_{\varrho(N-1)} & y_{\varrho i} \end{smallmatrix}$ respectively.

DEFINITION 2 *Let* $\Gamma = \{G_1, \ldots, G_p\}$ *be a collection of binary relations over* $\{0, \ldots, n\}$, *all having the same restriction*

$$G = \{(i,j) \in G_k \mid i \neq 0 \,\&\, j \neq 0\} \qquad (k = 1, \ldots, p)$$

to $\{1, \ldots, n\}$. *If this common restriction* G *differs from* G_k *for all* k, *then* $\{G\} \cup \Gamma$ *is said to be an* n-type. □

Next, in order to convey the intended meaning of a type t (which, so far, is just an instance of a data structure), we associate a formula Φ_t with t. Preliminary to defining Φ_t, we define the *merge* of a type.

DEFINITION 3 *Given an* n-*type* $t = \{G\} \cup \Gamma$, *let* $G_1, \ldots, G_p$ *be the members of* Γ *listed —for definiteness— in lexicographic order. The* merge *of* t *is the relation*

$$\mathcal{G}_t = G \cup \bigcup_{k=1}^{p} (\{(i, n+k) : (i,0) \in G_k\} \cup \{(n+k, i) : (0,i) \in G_k\}).$$

□

Let us denote by $\in_{ij}$ (leaving t implicit) the symbol $\in$ if $(i,j) \in \mathcal{G}_t$, and the symbol $\notin$ otherwise. Likewise introduce $\in_{ij}^k$, for $k = 1, \ldots, p$, referring this symbol to G_k instead of to $\mathcal{G}_t$. Let x_0 stand for the same variable which has been denoted by x till now.

$$\begin{aligned}\Phi_t \;=_{\text{Def}}\; & \left(\&_{i=1}^{n-1} \&_{j=i+1}^{n} x_i \neq x_j \,\&\, \&_{i,j=1}^{n} x_i \in_{ij} x_j\right) \,\&\\ & \left(\&_{k=1}^{p} \exists x_0 (\&_{j=1}^{n} x_0 \neq x_j \,\&\, \&_{i=0}^{n} (x_0 \in_{0i}^{k} x_i \,\&\, x_i \in_{i0}^{k} x_0))\right) \,\&\\ & \forall x_0 \left(\&_{j=1}^{n} x_0 \neq x_j \;\rightarrow\; \bigvee_{k=1}^{p} \&_{i=0}^{n} (x_0 \in_{0i}^{k} x_i \,\&\, x_i \in_{i0}^{k} x_0)\right).\end{aligned}$$

Establishing whether an n-type t is actually induced by an n-tuple of sets (see Definition 1) is a delicate matter, due to two reasons:

- Different universes of sets (i.e., interpretations based on a different domain or assigning a different meaning to $\in$) can be conceived[6] for the same theory T. It may turn out that t is induced by sets in some, but not in all, such universes.
- Depending on the theory T that is adopted, there are fewer or more universes of sets. It may turn out that t is induced by sets in some universes $\mathcal{U}$ of T, whereas no such $\mathcal{U}$ fulfills the axioms of a more demanding theory T′.

By Gödel's completeness theorem, the first situation arises when $\exists x_1 \ldots \exists x_n \, \Phi_t$ is neither provable nor refutable in T. The second situation arises when $\exists x_1 \ldots \exists x_n \, \Phi_t$, not refutable in T, is nevertheless refutable in T′. As will turn out, the first situation never takes place when T=$\mathcal{S}_{ER}$.

[6] For a better grasp of this point, consider the equation $x_1 = \{x_1\}$. This has no solutions in a universe modelling $\mathcal{S}_R$ —e.g. the classical von Neumann universe—. On the contrary, it will admit one, several, or infinitely many solutions in peculiar universes modelling $\mathcal{S}$ —cf. [Kri69].
Even the formula $\forall v \, v \notin x$ could be satisfied by an $x \neq \emptyset$, unless extensionality is postulated.

5 Filtering out useless types

We have devised, corresponding to each theory T $\in \{\mathcal{S}, \mathcal{S}_E, \mathcal{S}_R, \mathcal{S}_{ER}\}$, a boolean-valued procedure named T-*filter* that rejects —by answering *FALSE*— those n-types $t = \{G\} \cup \Gamma$ for which Φ_t is refutable in T. Denoting the T-filter by $\mathcal{F}$, $\mathcal{F}_E$, $\mathcal{F}_R$, $\mathcal{F}_{ER}$ in the respective cases, we have by definition:

$\mathcal{F}(t) \equiv_{\text{Def}}$ for every $J \subseteq \{1, \ldots, n\}$, there is a $k \in \{n+1, \ldots, n+|\Gamma|\}$ with $\{j : (j,k) \in \mathcal{G}_t\} = J$.

$\mathcal{F}_E(t) \equiv_{\text{Def}} \mathcal{F}(t)$ &, for $0 < i < j \leq n$, $\{h : (h,i) \in \mathcal{G}_t\} \neq \{h : (h,j) \in \mathcal{G}_t\}$.

$\mathcal{F}_R(t) \equiv_{\text{Def}} \mathcal{G}_t$ is an acyclic graph &, for every $J \subseteq \{1, \ldots, n\}$, there is a $k \in \{n+1, \ldots, n+|\Gamma|\}$ with $\{j : (j,k) \in \mathcal{G}_t\} = J$ and $\{j : (k,j) \in \mathcal{G}_t\} = \emptyset$.

$\mathcal{F}_{ER}(t) \equiv_{\text{Def}} \mathcal{F}_E(t)$ & $\mathcal{F}_R(t)$.

In order to justify $\mathcal{F}$, and to explain why this condition gets enriched inside $\mathcal{F}_R(t)$, we recall the theorem discussed at the beginning of —respectively the remark made at the end of— Section 2.

THEOREM 4 *Let* T $\in \{\mathcal{S}, \mathcal{S}_E, \mathcal{S}_R, \mathcal{S}_{ER}\}$ *and let* F *be the* T-*filter. When* F$(t) = FALSE$, *then* $\neg \Phi_t$ *is a theorem of* T. *On the other hand, the whole set of sentences*

$$\{\exists x_1 \cdots \exists x_n \, \Phi_t : n = 0, 1, 2, \ldots \text{ and } t \text{ is an } n\text{-type} \mid \mathtt{F}(t) = TRUE\}$$

is compatible with the axioms of T.

In the particular case T $= \mathcal{S}_{ER}$, *every sentence* $\exists x_1 \cdots \exists x_n \, \Phi_t$ *with* F$(t) = TRUE$ *is provable. Actually, one can find hereditarily finite sets*[7] *that satisfy* Φ_t *by, e.g., solving the system*

$$\left(\&_{i=1}^{n} x_i = \{x_j : (j,i) \in \mathcal{G}_t\}\right) \,\&\, \left(\&_{k=1}^{p} x_{n+k} = \{x_j : (j, n+k) \in \mathcal{G}_t\} \,\mathtt{with}\, \{k, n+p\}\right),$$

where p *is the number of graphs endowed with* $n+1$ *nodes in* t. □

This theorem indicates that when regularity, or extensionality, or both of them, are *not* assumed, a sentence of the form $\exists x_1 \cdots \exists x_n \, \Phi_t$, with t an n-type for some n, although not provable in general, cannot be refuted either. Even more is claimed: all sentences of this kind can be added *together*, with the status of new axioms, to the theory. The overall consistency of the theory will not be jeopardized by this extension.

In this manner one obtains three theories: a theory that systematically injures regularity, another one that systematically injures extensionality, and a third one that violates both regularity and extensionality in infinitely many ways. All three constitute non-standard (i.e. antithetic to ZF) set theories in which $\forall_n \exists$-completeness is restored. At least the first of these —closely related to Boffa's theory (cf. [Acz88])— appears to be worthy of study in its own right.

[7] A set is said to be *hereditarily finite* iff it can be denoted by a variable-free term which involves only the constructs $\emptyset$ and with. Among the hereditarily finite sets, one finds all natural numbers; for, in compliance with a classical trick due to von Neumann, each integer $M \geq 0$ is identified here with the set $\{0, \ldots, M-1\}$.

In Zermelo-Fraenkel set theory ZF the status of each n-type $t = \{G\} \cup \{G_1, \ldots, G_p\}$, as far as satisfiability or refutability is concerned, is exactly the same as in $\mathcal{S}_{ER}$. Hence in ZF, as well as in $\mathcal{S}_{ER}$, the following holds:

COROLLARY 5 *To every n-tuple $a_1, \ldots, a_n$ of distinct sets there corresponds an n-tuple $H_1, \ldots, H_n$ of hereditarily finite sets of rank $K(n)$ —where K is a specific elementarily computable function— that fulfills, via the replacement $x_1 \mapsto H_1, \ldots, x_n \mapsto H_n$, those same $\forall x\, p$-formulae which are true under the replacement $x_1 \mapsto a_1, \ldots, x_n \mapsto a_n$.* □

This result, which will be further clarified by the next section, certainly *cannot* generalize to formulae of the kind $\forall y \forall x\, q$, because the existence of infinite sets is expressible (cf. [PP89]) by means of a sentence of the form $\exists x_1 \exists x_2 \forall y \forall x\, q$, where q is unquantified and involves only the variables x_1, x_2, y, x, along with the constructs $\in$, $=$, $\neg$, $\&$.[8]

6 Decomposition of a formula $\forall x\, p$ and satisfiability detection

We now proceed to discuss a number of properties of the family $\mathcal{T}^{(n)}$ of all n-types. We begin by considering the properties that exclusively depend on predicate calculus with equality, without reference to any theory. First we observe that each t in $\mathcal{T}^{(n)}$ is, in a way, *fully informative* with respect to a formula $\forall x\, p$ of the kind described at the beginning of the preceding section. Mechanically detecting whether $\Phi_t \rightarrow \forall x p$ or $\Phi_t \rightarrow \neg\forall x p$ holds is indeed a very straightforward matter. (Notice that these two implications cannot hold together —of course the situation becomes more intriguing when there are axioms making it possible to refute Φ_t). Notice also that n-types are *exhaustive*, in the sense that $\bigvee_{t \in \mathcal{T}^{(n)}} \Phi_t$ holds. As a consequence, one has the *decomposability* of each formula $\forall x\, p$; as a matter of fact $\forall x\, p \leftrightarrow \bigvee_{t \in \mathcal{T}_p^{(n)}} \Phi_t$ holds, where the definition $\mathcal{T}_p^{(n)} = \{t \in \mathcal{T}^{(n)} | \Phi_t \rightarrow \forall x\, p \text{ holds}\}$ applies. This decomposition is irredundant (in the absence of axioms), because n-types *mutually exclude* each other; otherwise stated, $\neg(\Phi_{t_1} \& \Phi_{t_2})$ holds, provided t_1 and t_2, belonging to $\mathcal{T}^{(n)}$, are distinct.

Then come the properties of $\mathcal{T}^{(n)}$ that refer to set theories: n-types can be *filtered*, with respect to $\mathcal{S}$, as well as with respect to $\mathcal{S}_E$, $\mathcal{S}_R$, $\mathcal{S}_{ER}$, and ZF. Otherwise stated, with respect to each of these theories we are able to detect whether the formula Φ_t of an n-type t is refutable. The decomposition of each formula $\forall x\, p$ becomes lighter, as the refutable disjuncts are dropped in order to reinforce irredundancy. Quite nicely, the $\mathcal{S}_{ER}$-filter happens to be a direct combination of the $\mathcal{S}_E$-filter with the $\mathcal{S}_R$-filter.

Certainly, it would be preferable to read directly off p the disjuncts $\{\Phi_t : t \in \mathcal{T}_p^{(n)} | \mathcal{F}_{\mathtt{T}}(t) = \mathit{TRUE}\}$ of the decomposition of $\forall x\, p$ in the theory T, without having to generate the whole family of n-types in the first place. We have found that a bottom-up approach of this nature

[8] If regularity is postulated in conjunction with extensionality, one can choose as q the formula

$$x_1 \neq x_2 \;\&\; \&_{i=1}^{2}(x_i \notin x_{3-i} \& (x \in y \;\&\; y \in x_i \rightarrow y \in x_{3-i})) \& (y \in x_1 \;\&\; x \in x_2 \rightarrow y \in x \vee x \in y).$$

In the absence of regularity, the infinity statement becomes about twice as long.

is indeed viable, and proceed to give some details here, still assuming that p has the form (*) shown at the beginning of Section 4.

We initially bring p_* to disjunctive normal form $\bigvee_{i=1}^{m} p_i$, where each p_i is a conjunction of membership literals[9]. Let q_i be the conjunction of all literals in p_i that do not contain x, and let t_i be the conjunction of the remaining literals in p_i.

Let $q_{i1}, \ldots, q_{ih_i}$ be all maximal propositionally satisfiable conjunctions of literals extending q_i, in the variables $x_1, \ldots, x_n$. Likewise, let $t_{i1}, \ldots, t_{ik_i}$ be the maximal propositionally satisfiable conjunctions of literals extending t_i, with x as one side and one of $x, x_1, \ldots, x_n$ as the other side.

Any q_{ij}, taken together with an arbitrary nonempty subset $t_{ij_1}, \ldots, t_{ij_p}$ of $t_{i1}, \ldots, t_{ik_i}$, determines in a straightforward manner a type Φ_t. The disjunction of all Φ_t's obtainable in this fashion turns out to be the desired decomposition of p. Indeed, this follows from the availability of the three theorems $\exists x(\&_{i=1}^{n} x \neq x_i)$, $q_i \leftrightarrow \bigvee_{j=1}^{h_i} q_{ij}$, and $t_i \leftrightarrow \bigvee_{j=1}^{k_i} t_{ij}$ in each one, T, of our membership theories.

Acknowledgements. The remark that the decision problem relative to multilevel syllogistic with singleton reduces to the decision problem for Gogol's collection of $\forall x\, p$-formulae is due to A.Ferro and D.Cantone.

References

[Acz88] P. Aczel. *Non-well-founded sets.* Vol 14, Lecture Notes, Center for the Study of Language and Information, Stanford, 1988.

[Ble77] W.W. Bledsoe. Non-resolution theorem proving. *Artificial Intelligence*, 9,1–35, 1977.

[BLM*86] R. Boyer, E. Lusk, W. McCune, R. Overbeek, M. Stickel, and L. Wos. Set theory in first-order logic: Clauses for Gödel's axioms. *Journal of Automated Reasoning*, 2, 289–327, 1986.

[Bro78] F.M. Brown. Towards the automation of set theory and its logic. *Artificial Intelligence*, 10(3),281–316, 1978.

[CFO89] D. Cantone, A. Ferro, and E.G. Omodeo. *Computable set theory.* Oxford University Press, International Series of Monographs on Computer Science, 1989.

[COP89] D. Cantone, E.G. Omodeo, and A. Policriti. The automation of syllogistic. II. Optimization and complexity issues. *Journal of Automated Reasoning*, 1990.

[DF89] E.-E. Doberkat and D. Fox. *Software Prototyping mit SETL. Leitfäden und Monographien der Informatik*, B.G. Teubner Stuttgart, 1989.

[DLS89] V.T. Digricoli, J.J.Lu, V.S.Subrahmanian; And-or graphs applied to RUE resolution. In *Eleventh International Joint Conference of Artificial Intelligence, Proceedings Vol 1*, p.354–358, 1989.

[9] From now on, *literal* will mean membership literal with a variable on each side of $\in$ or $\notin$.

[FOS80] A. Ferro, E.G. Omodeo, and J.T. Schwartz. Decision procedures for elementary sublanguages of set theory. I. Multi-level syllogistic and some extensions. *Communications on Pure and Applied Mathematics*, 33,599–608, 1980.

[Fri63] J. Friedman. A computer program for a solvable case of the decision problem. *J.ACM*, 10,348–356, 1963.

[Gog78] D. Gogol. The $\forall_n\exists$-completeness of Zermelo-Fraenkel set theory. *Zeitschr. f. math. Logik und Grundlagen d. Math.*, 1978.

[Kri69] J. K. Krivine. *Theorie axiomatique des ensembles.* Presses Universitaires de France, 1969.

[LS80] D.W. Loveland and R.E. Shostak. Simplifying interpreted formulas. In *Proceedings of the 5th conference on Automated Deduction*, Vol 87, Lecture Notes in Computer Science, p.97–109, Springer-Verlag, 1980.

[Pas78] D. Pastre. Automatic theorem proving in set theory. *Artificial Intelligence*, 10( ),1–27, 1978.

[PP88] F. Parlamento and A. Policriti. Decision procedures for elementary sublanguages of set theory. IX. Unsolvability of the decision problem for a restricted subclass of the Δ_0-formulas in set theory. *Communications on Pure and Applied Mathematics*, 41,221–251, 1988.

[PP89] F. Parlamento and A. Policriti. Note on "The logically simplest form of the infinity axiom". *Proceedings of the AMS*, 1990.

[Rob67] J.A. Robinson. A review of automatic theorem-proving. *Proc. Symp. Appl. Math.*, Vol 19, Amer. Math. Soc., p.1–18, Providence, R.I., 1967.

Global, local and weak graph properties for normal logic programs

Agostino Cortesi - Gilberto File'

Dip. di Matematica Pura e Applicata
Universita' di Padova
via Belzoni 7
I-35131 PADOVA (Italy)

Abstract *The aim of this work is to study the relationship among some classes of normal programs which have consistent completion. This systematic study shows a few basic properties, that are sufficient for allowing the computation of a model of the completion, and the links with the satisfiability of a Circumscription formula. In this investigation it was natural to consider a class of programs, called weakly call-consistent, that is an extension of the weakly stratified class.*

Introduction

The consistency of the Clark completion is important because it is necessary for the completeness of SLDNF resolution and of Negation as Failure. The aim of this work is to give a general scheme for the classes of normal programs with consistent completion till now defined, namely definite, stratified, call-consistent, locally stratified, locally call-consistent and weakly stratified.

In **[ABW]** it is shown that every stratified program has consistent completion: for the proof a model of the completion is defined in an iterative way on the strata of the program. Later Przymusinski **[Prz1]** has shown that the model of **[ABW]** is perfect, minimal and also that it satisfies a priorized circumscription formula **[McC, Lif].** A similar construction can be used to show that also call-consistent programs have consistent completion: the model is built iteratively on the equivalence classes of the global dependency graph (see section 1), instead of the strata. This result was first shown by **[Kun]** using 3-valued logic. We prefer to recall a construction given in **[Bar]** that marries the approach of **[Kun]** and the Fitting's notion of partial model **[Fit]** and that seems to us very illuminating of the main ideas behind the result. The model constructed by this technique will be called below a KB-model. It is also interesting to observe that also KB-models satisfy a priorized circumscription formula **[CF]**.

Here we want to stress a fundamental point of all these constructions: the program is divided in pieces that are partially ordered. As long as one considers finite programs this order is obviously well-founded. However, when one wants to extend these classes to the local level, this fact is no longer obvious. To define the locally stratified **[Prz1]** and locally call-consistent programs **[Sat, Cav]**, one must consider all the ground instances of the original programs obtaining in this way infinite programs. Therefore, for these classes, in addition to requiring that the program can be divided into

partially ordered strata or equivalence classes (with certain properties), one must explicitly require that the partial order is well founded. Only the class of the locally semi-strict programs **[Sat]** does not require this property and, in fact, the problem of whether the completion of such programs are consistent is still open.

In order to extend the class of locally stratified programs, Przymusinski introduced the notions of weak stratification and weakly perfect model: the idea is to modify step by step the program in order to remove all "irrelevant" relations in the local dependency graph **[Prz2]**. Both perfect models of locally stratified programs and weakly perfect models of weakly stratified programs satisfy the priorized circumscription formula **[Prz1, Prz2]**.

Summarizing, we point out three main concepts at the basis of the definitions of all the classes of programs with consistent completion:
- the iterative construction of a KB-model;
- the requirement of a well-founded order that guarantees the extension to the local level;
- the Davis Putnam rule that enables a further extension of the classes.

We have organized and compared all existing classes w.r.t these three notions. To this end the diagram of *Fig 1* summarizes the result of this analysis (in this diagram the arrows indicate inclusions). Notice that for completing the schema it was natural to introduce the new class of weakly call-consistent programs, that extends properly the weakly stratified class. The completions of such programs are consistent, as shown in **[CF]**.

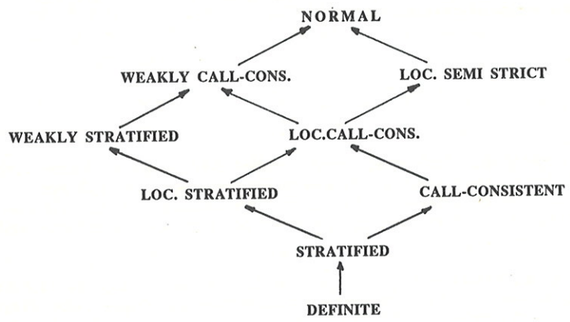


Figure 1: The lattice of normal logic programs

1. Preliminaries

Let L be a language, consisting of a set of variable, predicate and function symbols. The notions of term, formula, literal, clause, program, completion... are defined in the usual way.

PRED is the set of L-predicate symbols.

For every subset Q of *PRED*, $P(Q)$ denotes the set of all clauses in P whose heads contain a predicate of Q.

Let $p,q \in PRED$. Let us consider the *signed dependency relation* defined by: $p <_+ q$ (resp $p <_- q$) iff there exists a clause of P: $q(...) \leftarrow r_1(...), ..., r_m(...), \neg s_1(...), ..., \neg s_n(...)$ with $p = r_i$ (resp $p = s_j$). We will always refer to its reflexive and transitive closure, i.e.:

$p <_{+1} p$ and

if $p <_i q$ and $q <_j r$ then $p <_{i*j} r$, where * denotes multiplication.

We will call *global dependency graph* the directed graph associated to P w.r.t. the dependency relation $<_j$.

Likewise, we will call *local dependency graph* the directed graph associated to the (infinite) program P' of all the ground instances of the program P, w.r.t. the dependency relation $<_j$ in P'.

Notice that $p <_{-1} q$ if and only if in the global (resp. local) dependency graph associated to the program P (resp.P') there exists a path from p to q with an odd number of negative edges.

Let $p,q \in PRED$, $p \approx q$ if and only if $p < q$ and $q < p$.

Let [p] be an equivalence class w.r.t. the equivalence relation $\approx$. Let q be a selected element of [p]. We call q a *representative* of the class [p].

Let $p \in PRED$, p of arity m. Let $x=(x_1, ...,x_m)$ be an m-tuple of variables in L and let $q_1, ...,q_n$ be all predicates following p in the dependency graph (i.e. $p < q$). The *priorized circumscription* of the predicate p in the program P is the following second-order formula **[McC, Lif]**:

$Circ_p(P;\{q:p <q\}) = P(p, q_1, ...,q_n) \wedge \neg\exists\, x, p', q_1', ..., q_n' \, [P(p',q_1', ...,q_n') \wedge p(x) \wedge \neg p'(x)]$.

2. Global properties: stratification and call-consistency

Looking at the features of the global dependency graph of a normal program P, two important classes have been defined **[ABW, Prz1, Cav]**: *stratified and call-consistent programs*.

A logic program P is *stratified* if there exists a stratification of $PRED=D_1 \cup ... \cup D_h$ such that for every clause $p(...) \leftarrow q_1(...),...,q_n(...),\neg r_1(...),...,\neg r_m(...)$ *in* $P(D_k)$

i) all predicate symbols $q_1,...,q_n$ are in $\cup_{j\leq k} D_j$

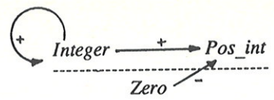
ii) all predicate symbols $r_1,...,r_m$ are in $\cup_{j<k} D_j$.

We can note that the global graph for a stratified program can be cut in a finite number of strata such that in none there is a cycle with negative edges.

example 1

Pos_int (x) ← *Integer (x),* ¬ *Zero (x).*
Integer (s(x)) ← *Integer (x).*
Zero (0).
Integer (0).

This program is stratified but not definite. In the dependency graph associated a stratification is shown:



In **[Prz1]** and **[ABW]** a semantical characterization of the stratified programs is given: *the perfect model semantics*.

Let $\mathfrak{M}$ and $\mathfrak{N}$ be models of a logic program P.
We say that $\mathfrak{N}$ is *preferable* to $\mathfrak{M}$ iff for every ground atom A in $\mathfrak{N} \setminus \mathfrak{M}$ there exists a ground atom B in $\mathfrak{M} \setminus \mathfrak{N}$ such that $A > B$.

A model $\mathfrak{M}$ is *perfect* if there are no models preferable to $\mathfrak{M}$.

Theorem 1 [ABW, Prz1, Lif]
A stratified program has a (unique) perfect model.

The computation of such a perfect model is given bottom-up, by iterating the immediate consequence map T_P on each stratum and finally by taking the set union.

Let A be a L-interpretation. We will use $B_{PRED,A}$ to denote the set $\{p(a_1,...,a_n)$: p is a n-ary predicate symbol and $a_1,...,a_n$ in the domain of A$\}$.

Theorem 2 [ABW]
Let P be a program, A a preinterpretation , $I \subset B_{PRED,A}$. Then $A \cup \{I\}$, together with an interpretation of "=" that satisfies the equality theory of comp(P), is a model of comp(P) iff I is a fixed point for the map $T_{P,A}$ defined as follows:
$T_{P,A}(I) = \{ D : D \leftarrow B_1, ..., B_r, \neg C_1, ..., \neg C_s$ *is a closed instance on A of a clause in P with* $B_i \in I$ *and* $C_j \notin I$, *for every* $1 \le i \le r$, $1 \le j \le s \}$.

Corollary: *The perfect model for a stratified program P is also a model for the completion of P.*

The computation of a perfect model for a stratified program follows the global dependency graph. It attempts the minimality of the extension of each predicate as much as it is possible by taking fixed all previous predicates and by taking free all following predicates w.r.t. the dependency order, as shown by this theorem:

Theorem 3 [Prz1]
The perfect model for a stratified program P satisfies the priorized circumscription formula $\cup_{p \in PRED} \mathrm{Circ}_p (P ; \{q : p < q\})$.

A more general class of normal programs whith consistent completion was defined **[Sat, Cav]**: *the call-consistent programs*.

A program P is *call-consistent* iff for no predicate symbol p, $p <_{-1} p$.

It is easy to prove that the following conditions are equivalent:
i) P is call-consistent.
ii) for $p,q \in$ PRED it never holds $p \approx q$, $p <_{+1} q$, $q <_{-1} p$.
iii) There exists a mapping *lev* such that for every predicate $p,q \in$ PRED , if p depends on q then $lev(p) \ge lev(q)$ and if p depends both positively and negatively on q then $lev(p) > lev(q)$.

↳ wide
$q <_{+1} p \Rightarrow S(q) \le S(p)$
$p <_{+1} q$ e $q <_{-1} p \Rightarrow S(q) < S(p)$

Then the graph characterization of the call-consistent programs is the following: in every strong connected component there is no cycle with an odd number of negatives edges.

Remark: *All stratified programs are call-consistent.*

The inverse is false, as shown by the following program, which is call-consistent but not stratified:

example 2 $p \leftarrow \neg q.$ $q \leftarrow \neg p.$ **p** ⇄ **q**

The computation of a model for the completion of a call-consistent program was shown by Kunen **[Kun]** with a 3-valued logic.

Theorem 4 [Kun, Bar]
Let P be a call-consistent logic program. There exists a model for the Clark's completion of P , i.e. comp(P) is consistent.

Here we consider the proof given in **[Bar]**, that relates Kunen's approach and Fitting's notion of partial model **[Fit]**. The main idea is to construct the model by induction on the dependency order, by expanding step by step the model to an upper minimal strongly connected component. At each step two subsets of B_P are computed: all instances that have to be true in the model (**I**) and all instances that have to be false (**J**), for every predicate in the given equivalence class; moreover, to decide the truth value of all predicate instances still "undecided", select a class-representative p : for every q in the class and for every h-tuple $(t_1,...,t_h)$ in the Herbrand Universe, set:

$q(t_1,...,t_h)$ true iff $q(t_1,...,t_h) \in \mathbf{I}$ or $q(t_1,...,t_h) \notin \mathbf{J}$ and $q <_{-1} p$.

These models are named *KB models*. Moreover if $\psi_1,...,\psi_n$ are the selected representatives of all the equivalence classes, say $[\psi_1],...,[\psi_n]$, the *KB-model* will be denoted by $KB_{\psi_1,...,\psi_n}$.

Theorem 5 [CF]
Let P be a stratified program. The (unique) KB-model for comp(P) is perfect.

Remark *Generally the KB-models for call-consistent programs are not perfect, as shown by example 2: this program has two different KB-models, one for every selected representative in the unique equivalence class: M = {p} and N = {q}. The first one is preferable to the second one and viceversa. Then no one is perfect.*

Theorem 2 creates a link between the perfect model for a stratified program P and the satisfiability of the priorized circumscription formula on all predicates in P.
A similar result can be shown for call-consistent programs by considering the priorized circumscription w.r.t. a set of predicates S such that S contains exactly one representative for each equivalence class.

Theorem 6 [CF]
Let P be a call-consistent program. The model $KB_{\psi_1,...,\psi_n}$ *for the Clark's completion of P satisfies the priorized circumscription formula restricted on* $\psi_1,...,\psi_n$, *that is:*

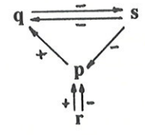
$$\cup_{\psi_i} \mathrm{Circ}_{\psi_i} (P; \{q: \psi_i < q\}).$$

Note that every equivalence class $[\psi]$ is separable on two disjoint subsets $[\psi]^+$, $[\psi]^-$ such that : if , $j \in \{+,-\}$ $p,q \in [\psi]^j$ then $p <_{+1} q$ and $q <_{+1} p$. Then it is easy to see that every representative in $[\psi]^+$ (or in $[\psi]^-$) gives the same expansion, and so it holds the following theorem:

Theorem 7 [CF]
Let P be a call-consistent program. Let n be the number of the equivalence classes w.r.t. the equivalence relation ≈. Let $m \leq n$ be the number of the classes without negation. The KB-models for the Clark's completion of P are exactly 2^{n-m}.

example 3:

$q(x) \leftarrow \neg s(x), p(x).$
$s(x) \leftarrow \neg q(x).$
$p(x) \leftarrow \neg s(x), r(x).$
$p(x) \leftarrow \neg r(x).$
$p(S(0)).\quad r(0).$



The equivalence classes for P are [r] and [p,q,s]. Let us select r and p.
$KB_{r,p}$= {r(o), p(n), s(m): $0 \leq m, 0<n$} satisfies the circumscription formula $Circ_r(P; \{r,p,q,s\})$ $\cup Circ_p(P; \{p,q,s\})$. Moreover we can see that the KB-models for P are exactly $2^{(2-1)} = 2$, i.e. $KB_{r,s}$ and $KB_{r,p} = KB_{r,q}$

3. Local properties.

Till now we have seen two classes of programs that are guaranteed to have consistent completion: the stratified and the call-consistent programs.
The proofs of the consistency are based on two facts:

1. the existence of a well founded partial order on the dependency graph
2. the use of this order for constructing KB-models.

Other extended classes that have consistent completion have been defined **[Cav, Prz1, Prz2, Sat]**. They also use appropriate extensions of the above 2 facts. In what follows, the definitions of these extended classes will be presented in a systematic way, based on the above 2 points, in order to clarify the relationship among them.

The first extension consists in requiring the existence of a well founded order on the predicate instance level, called *local level:*. The main problem here is that while at the global level the dependency graph was finite, at the local level we can meet infinite dependency chains. Then a strong condition on the order has to be imposed to ensure the computability of a completion model.

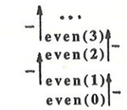
A logic program P is *locally stratified* if it is possible to decompose the Herbrand basis of P into disjoint sets, called strata $H_0, H_1,...,H_\alpha,...$ where $\alpha<\gamma$ and γ is a countable ordinal, so that for every ground clause $A \leftarrow B_1, ...,B_r,\neg D_1, ...,\neg D_k$ if A belongs to H_k then

i) all positive premises B_i belong to $\cup\{H_i: i \leq k\}$

ii) all negative premises D_j belong to $\cup\{H_i: i < k\}$.

example 4:

P: *even(0).*
even(s(x)) $\leftarrow \neg$ even(x).



This program is locally stratified but not stratified

Przymusinski in **[Prz1]** shows that *every locally stratified program has (exactly) one perfect model*. From the construction of the model one can easily see that it's also a model of the Clark's completion. Perfect models of locally stratified programs satisfy the priorized circumscription formula **[Prz1, Prz2]**.

Consider a program consisting only of ground clauses and let $H_0, H_1,...,H_\alpha,...$ be a decomposition of the Herbrand base of *P*. With P_i we denote the program obtained by taking the clauses $A \leftarrow \alpha$ of *P* such that *A* belongs to H_i and dropping from α all literals that do not belong to H_i. If *P* is locally stratified w.r.t. the given decomposition, then *P* is obviously a definite program (without negation). One may wonder whether a class larger than that of locally stratified programs could be obtained if one requires for any program *P* the existence of a decomposition *D* of the Herbrand Base of *P* such that each program P_i is stratified (instead of definite as before). It is easy to see that it is not the case: the decomposition *D* can be refined obtaining a new decomposition w.r.t. which *P* is locally stratified.

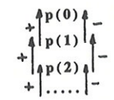
Call-consistency also has been extended at the local level: such extension can be defined in two ways **[Cav, Sat, Bar]**:

A program *P* is *locally call-consistent* if it has a local mapping *lev* such that for every atoms *A,B* in the Herbrand Basis, if *A* depends on *B* then $lev(B) \leq lev(A)$ and if *A* depends both positively and negatively on *B* then $lev(B)<lev(A)$.

A program *P* is *locally semi-strict* **[Cav]** (negative cycle free in **[Sat]**) if for no *A* in the Herbrand Base of *P* $A <_{-1} A$.

Obviously, for the locally call-consistent class, *lev* guarantees that the dependency relation on the predicate instance is well founded, whereas this is not the case for the locally semi-strict programs. In fact, one can easily see that, at the local level, these two classes are different:

example 5 **[Cav]**. *P:* *p(0).*
p(x) $\leftarrow$ p(s(x)).
p(x) $\leftarrow \neg$ p(s(x)).



This program is locally semi-strict but not locally call-consistent.

As shown in **[Sat]** the completion of a locally call-consistent program is consistent, whereas for the locally semi-strict programs the question is still open **[Cav]**.

4. Weak properties.

Przymusinski in **[Prz2]** extended the locally stratified class to the *weakly stratified* programs. The main idea is to remove all "irrelevant" relations in the local dependency graph, modifying the program by the Davis Putnam rule.

Let P be a program such that the partial order < on the strongly connected components in the dependency graph is well founded.

The *bottom stratum* is the union of all minimal components of P (i.e.: $S(P) = \cup\{C: C \text{ is a minimal component in the graph of } P\}$), and the *bottom layer* $L(P)$ is the corresponding subprogram of P, i.e. the set of all clauses from P whose heads belong to $S(P)$.

Given M a subset of $S(P)$, for an atom A in $S(P)$ we will say that *A is true in M* iff A is in M. By a *reduction modulo M* we mean a new program P/M obtained from P by performing the following two reductions (Davis Putnam rule):

- removing from P all clauses already satisfied in M.
- removing from all the remaining clauses those premises which are satisfied in the model M.

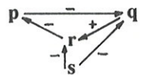
Then we define a recursive operation which leads to the construction of a weakly partial perfect model: define P_1 to be P itself, $S_1 = S(P_1)$, $L_1 = L(P_1)$ and let M_1 be the least model of L_1 if it exists. Having defined P_k, S_k, L_k and M_k for some k $(0 \le k)$, we define $P_{k+1} = P_k / M_k$ and proceed as follows:

- if S_{k+1} is empty then the construction stops and $M_P = M_1 \cup M_2 \cup ... \cup M_k$ is defined to be the *weakly perfect model M_P of P*.
- otherwise, let $S_{k+1} = S(P_{k+1})$ and $L_{k+1} = L(P_{k+1})$ and let M_{k+1} be the least model of L_{k+1}.

A logic program P is *weakly stratified* if all layers L_i are *positive logic programs*. Thus, obviously, every weakly stratified program has a unique weakly perfect model.

A simple example may help the intuition.

example 6 $P =$ $s \leftarrow .$
$q \leftarrow \neg s, \neg p.$
$r \leftarrow \neg s, q.$
$p \leftarrow \neg r.$



$S_0 = \{s\}$ $M_1 = \emptyset$
$L_0 = \{s \leftarrow .\}$ $P_2 = P_1 / M_1 = \{p \leftarrow .\}$
$M_0 = \{s\}$ $S_2 = \{p\}$
$P_1 = P_0 / M_0 = \{p \leftarrow \neg r.\}$ $L_2 = \{p \leftarrow .\}$
$S_1 = \{r\}$ $M_2 = \{p\}$
$L_1 = \emptyset$ $P_3 = P_2 / M_2 = \emptyset$

$M_P = M_0 \cup M_1 \cup M_2 = \{s,p\}$.

This program is weakly-stratified but not locally stratified

Also weakly perfect models of weakly stratified programs satisfy the priorized circumscription formula **[Prz2]**.

Let us summarize. Till now we have seen two ways of assuring a well founded order: stratification and call-consistency. Both these ways have been considered both at the predicate symbol level and at the local level. Finally, Przymusinski introduced the idea of simplifying the program (by the Putnam Davis method) while iteratively constructing its model, but he applied it only to the definite approach and not to the stratification and call-consistency approach. It is easy to see, in fact, that the previous construction is sound even when all layers L_i are stratified or call-consistent, since we know the way to construct step by step a perfect or KB-model for these programs.

As observed before for the locally stratified programs, requiring that all layers are stratified logic programs instead of definite ones does not result in a larger class of programs. At the constrary the call-consistent property gives a larger class:

A logic program is *weakly call-consistent* if all layers are call-consistent logic programs.

Clearly, every weakly call-consistent program has a (generally non unique) weakly-KB-model.

example 7. $p \leftarrow \neg q.$ $q \leftarrow \neg p.$

This program is weakly call-consistent but not weakly stratified:

$S_0 = \{p, q\}$
$L_0 = \{p \leftarrow \neg q.\ q \leftarrow \neg p.\}$
$M_0 = \{p\}$ $M'_0 = \{q\}$
$P_1 = P_0 / M_0 = \emptyset$ $P'_1 = P_0 / M'_0 = \emptyset$
$M_P = \{p\}$ $M'_P = \{q\}$

Theorem 8. [CF]
Every locally call-consistent program P is weakly call-consistent.

Remark that generally the inverse is false, as shown by the following **[Prz2]**:

example 8. $p(1,2).$
$q(x) \leftarrow p(x.y), \neg q(y).$

The new class satisfies the expected consistency property.

Theorem 9 [CF]
Let P be a weakly call-consistent program. The weakly-KB-models for P satisfy the Clark completion of P.

5. References

[ABW] Apt,K., Blair,H. and Walker,A., *Towards a theory of declarative knowledge*, in Foundations of deductive databases and logic programming, J.Minker ed., Morgan Kaufmann, Los Altos, CA, 1987.

[Bar] Baratella,S., *Models of Clark's completion for some classes of logic programs*, 1989, unpublished.

[Cav] Cavedon,L., *A completeness theorem for SLDNF-resolution*, Technical report 88/17, Dept. of Comp. Sci. Univ. of Melbourne, 1988.

[CF] Cortesi,A., File',G. *Classes of Programs with consistent completion*, unpublished 1989.

[Fit] Fitting,M, *Partial models and logic programming*, Theoretical Computer Science 48, 1986

[Kun] Kunen,K., *Signed data dependencies in logic programs*, Computer Sciences Technical Report #719, Univ. of Wisconsis, 1987.

[Lif] Lifschitz,W., *On the declarative semantics of logic programs with negation*, in Foundations of deductive databases and logic programming, J.Minker ed., Morgan Kaufmann, Los Altos, CA, 1987.

[Llo] Lloyd,J.W., *Foundations of logic programming*, second edition, Springer, New York, 1987.

[McC] McCarthy,J., *Applications of circumscription to formalizing common sense knowledge*, Artificial Intelligence 28 (1), 1986.

[Prz1] Przymusinski,T.C., *On the declarative semantics of deductive databases and logic programs*, in Foundations of deductive databases and logic programming, J.Minker ed., Morgan Kaufmann, Los Altos, CA, 1987.

[Prz2] Przymusinski,T.C. *Weakly perfect model semantics for logic programming*, Proc. V Conf.on Logic Programming Seattle , ed.Kowalski, 1988.

[Sat] Sato,T., *Completed logic programs and their consistency* , to appear in Journal of Logic Programming, 1988.

A Theory for Modeling the Synchronization Mechanisms of Concurrent Logic Languages

Catuscia Palamidessi

Dipartimento di Informatica, Corso Italia, 40, 56125 Pisa ITALY

1. Introduction.

Concurrent Logic Languages offer a high-level computational model that lends itself to a wide range of concurrent programming techniques. They are based on a subset of the First Order Logic, namely the Horn Clause Logic (HCL) [vanEmden76, Lloyd87, Apt88]. The main advantage of logic languages is that programs are sets of assertions which have a declarative reading. In other words, they can be seen as the description of the problem to be solved, and they can be understood without any reference to the behaviour of any particular machine. An other advantage is the possibility of representing data structures as logical terms and manipulating them by using unification. Finally, these languages seem, by their nature, quite suitable to express the parallelism implicit in certain problems. In particular, the atoms in a goal can naturally be interpreted as processes, and the shared variables as communication channels. However, pure HCL languages are not expressive enough to model the actual aspects of concurrent systems, such as real time, synchronization, nondeterminism-control.

Most of the Concurrent Logic Languages extend HCL by adding some constructs for explicit concurrency. In particular, constructs for guarded-nondeterminism and for synchronization. The proposals for controlling nondeterminism are based essentially on the introduction of the commit operator and of the guarded-clause. On the other side, there are many proposals concerning the synchronization mechanisms. The most popular one is based on the idea of constraining the unification (input-constraints). In practice, some of the variables occurring in a process can be prevented from getting bound during the unification preceding a computation step. In this way, the process in which such a variable occur can be forced to wait (suspension rule) until other processes in the goal have made available the required bindings. In this way, the shared variables can be seen as directed channels, and the suspension rule provides a synchronization mechanism. This class of languages includes Concurrent Prolog (CP) [Shapiro83, Shapiro86, Shapiro87], Guarded Horn Clauses (GHC) [Ueda85, Ueda86] and PARLOG [Clark85, Clark86, Gregory87].

It has often been argued that the additional features of Concurrent Logic Languages heavily affect the clearness and the semantics purity of HCL. The main problem is not the commit operator. In fact, the presence of commit does not change the success set [Lloyd87] of a logic program. On the contrary, the input-constraints usually restrict the number of successful computations. More in general, the declarative reading of a predicate definition is lost. For instance, the invertibility of the arguments does not hold anymore.

The first step to restore the declarative clearness of logic languages is to give a declarative definition of the input-constrained unification. Originally, this notion has been defined for CP by showing how to modify the unification algorithm, when the read-only variables are present [Shapiro83]. More recently, a declarative definition of the input-constrained unification (for CP) has been given in [Shapiro87, Szoke88]. This proposal, however, is not yet completely satisfactory. We discuss it in section 5. For the languages GHC and PARLOG, in the original papers the input constraints are defined only informally. A more formal definition has been given in [Shapiro88]. These constraints are less complicated than in CP, and such a definition could suffice for a clear understanding of the operational semantics. However, for dealing with the declarative semantics, a logical theory of constrained unification is needed. An attempt in this direction has been made in [Palamidessi88, deBoer89]. However this definition is too complicate, and it still requires to modify the unification algorithm, in order to compute the most general unifier.

The theory of constrained unification described in this paper allows to model the synchronization mechanisms of many languages. Moreover, we show that the difference between these languages just rely on few parameters in this theory. Then, we can conceive a general framework in which to describe the common properties of these languages, by abstracting from the specific synchronization mechanisms. A nice consequence of this is discussed in [Ciancarini89]. It turns out that we can distinguish between two component in the Concurrent Logic Programming paradigm: the *control* and the *synchronization*. So, by the equation

Concurrent Logic Language = control + synchronization

it follows that all the languages mentioned above can be seen as instances of a general schema in which the synchronization represents the (only) parameter.

An other interesting approach is based on Constraint Logic Programming [Jaffar87]. This approach, very elegant from a semantical point of view, was firstly investigated by Maher [Maher87] as an application of CLP to the specification of synchronization mechanisms. Successively, Saraswat [Saraswat89] has defined a family of languages based on this paradigm and he has shown that most of the concurrent logic languages can be subsumed in it. Furthermore, he has showed that the difference between the elements of this family mainly relies on the different nature of the constraints. Then, also here it is possible to delineate a general schema in which the semantic properties are defined in a way that is parametric w.r.t. the synchronization mechanisms. A special class of constraints has been used in the definition of NGHC [Falaschi89] and ccGHC [Gabbrielli89]. In these languages a special technique for constraints manipulation has been developed in order to define correct unfolding rules.

The paper is organized as follows. In section 2 we introduce the class of Concurrent Logic Languages that include CP, GHC and PARLOG. In section 3 we give the basic definition of a constrained-unification general theory. In section 4 we show how this theory applies to the particular cases of CP, GHC and PARLOG. Finally, in section 5 we compare our theory with the ones mentioned above.

2. The concurrent logic languages based on constraints on unification.

In this section we introduce briefly the class of concurrent logic languages whose synchronization mechanism is based on input-mode constraints. We introduce first the common structure these languages have, and then we present in more details the particular cases of CP, GHC, and PARLOG.

The alphabet consists of a set P of *predicate* symbols $p, q, r,...$, a set C of *constructor* symbols, $a, b, c,...$, $f, g, h,...$, and a set V of *variables*, $x, y, z,...$. The basic construct is the *guarded clause*:

$$A \leftarrow G_1,...,G_m \mid B_1,...,B_n.$$

where $A, G_1,...,G_m, B_1,...,B_n$ are *atoms*, namely objects of the form $p(t_1,...,t_k)$, where $t_1,...,t_k$ are *terms* built on C and V. A is called the *head* of the clause, $G_1,...,G_m$ is called the *guard part*, $B_1,...,B_n$ is called the *body part*. The symbol "|" is called *commit operator*. A *program* is a finite set of guarded clauses. A *goal* is a construct of the form

$$\leftarrow B_1,...,B_k.$$

where $B_1,...,B_k$ are atoms.

According to the *process interpretation* [Shapiro83, Levi85] a goal can be seen as a network of processes (the atoms) communicating via the shared variables (seen as channels). Some constraints (input constraints) can be specified on the clauses and goals. Their role is essentially to prevent the use of a certain clause, during the computation of a process. CP, GHC and PARLOG use different kinds of input-constraints. We will consider them later.

We introduce now the computational model. The actual computational models of CP, GHC and PARLOG slightly differ from each other, and from the one we present here. However, the differences are not so relevant and they do not concern the main features of the synchronization mechanisms.

The execution of a process B_i is based on the *derivation step*. The clauses of the program are tried in parallel. They are renamed, in order to avoid the clash of variables. The attempt to use a clause $A \leftarrow G_1,...,G_m \mid B'_1,...,B'_n$ can give one of the following results:

- *fail*, if A and B_i are not unifiable (i.e. they cannot become identical by a substitution variables-terms), or the guard part (seen as a goal) *fails* (see below).
- *success*, if A and B_i are unifiable, the guard part (seen as a goal) *succeeds* (see below), and the input-constraints are satisfied,
- *suspend*, if A and B_i are unifiable, the guard part is verified, but the input-constraints are not satisfied.

If all the clauses fail, then B_i (and the whole goal) *fails*. If no clauses succeed, and some of them suspend, then B_i *suspends*. If some of the clauses succeeds, then one of them is nondeterministically selected, say $A \leftarrow G_1,...,G_m \mid B'_1,...,B'_n$, and all the other attempts are discharged. Then, the derivation step takes place: B_i is replaced by $(B'_1,...,B'_n)\vartheta$, where ϑ is the composition of the most general unifier of A and B_i, and of the substitution computed during the derivation of $G_1,...,G_m$. Moreover, ϑ is applied to the other processes $B_1,...,B_{i-1}, B_{i+1},...,B_k$ (bindings-sending). The derived goal is then

$$\leftarrow (B_1,...,B_{i-1}, B'_1,...,B'_n, B_{i+1},...,B_k)\vartheta,$$

and ϑ is the *answer substitution* computed by this derivation step. After a derivation step, the processes in the goal that were suspended may be resumed. Indeed, the bindings-sending can eliminate the causes of suspension. Namely, some input-constraints can become satisfiable. The original goal *succeeds*, with computed answer substitution σ, if the empty goal is derived, and σ is the composition of all the substitutions computed during the derivation steps.

The input-constraints are the synchronization mechanism of this class of languages. Indeed, a process can be forced to wait until other processes have produced the necessary bindings on the shared variables. In the rest of this section we discuss the various kind of constraints adopted in the specific languages.

2.1. Concurrent Prolog.

In CP the variables in the atoms can be annotated by a symbol "?". In this case, they are called *read-only* variables. The input-constraints of CP specify that the read-only variables must not get instantiated, neither during the unification with the head of a clause, nor during the valuation of the guard. It is necessary a definition "ad hoc" for the application of a substitution. The application of a substitution ϑ to a read-only variable $x?$ gives:

- $x?$ if ϑ does not bind x,
- $y?$ if ϑ binds x to a variable y,
- t if ϑ binds x to a not-variable term t.

The application of a substitution to a term extends naturally.

Example 2.1. Merging lists. The following program defines in CP a process that performs the merge on lists. *nil* is the empty list, and *cons(x,y)* is the list obtained by prefixing a list y with an element x.

1) $merge(cons(x,y), y', cons(x,z)) \leftarrow \mid merge(y?, y', z)$.
2) $merge(y,cons(x, y'), cons(x,z)) \leftarrow \mid merge(y, y'?, z)$.
3) $merge(nil, y, y) \leftarrow \mid$.
4) $merge(y, nil, y) \leftarrow \mid$.

Consider two processes p and q that produce lists of elements a's and b' s respectively, as defined by the following clauses

5) $p(cons(a,y_1)) \leftarrow \mid p(y_1)$.
6) $p(nil) \leftarrow \mid$.
7) $q(cons(b,y_2)) \leftarrow \mid q(y_2)$.
8) $q(nil) \leftarrow \mid$.

Consider the goal

$\leftarrow p(y_3), q(y'_3), merge(y_3?, y'_3?, z_3)$.

The process *merge* is forced to wait until some bindings are produced on the variables y_3 or y'_3. Indeed, all the clauses for *merge* suspend. Assume that p performs a step by using the clause 5. Then the substitution $\{y_3/cons(a,y_1)\}$ is computed and the goal becomes

$\leftarrow p(y_1), q(y'_3), merge(cons(a,y_1), y'_3?, z_3)$

The process *merge* may now perform a step by using the clause 1 (the computed substitution would be $\{x/a, y_1/y, y'/y'_3?, z_3/cons(a,z)\}$). The clauses 2 and 4 still suspend, whilst the clause 3 fails. Assume that first q performs a step, by using the clause 7. Then the substitution $\{y'_3/cons(b,y_2)\}$ is computed and the goal becomes

$\leftarrow p(y_1), q(y_2), merge(cons(a,y_1), cons(b,y_2), z_3)$

Now, both the clauses 1 and 2 can be used for the process *merge*. Assume that the clause 1 is selected. Then, the substitution $\{x/a, y_1/y, y'/cons(b,y_2), z_3/cons(a,z)\}$ is computed and the goal becomes

$\leftarrow p(y), q(y_2), merge(y?,cons(b,y_2),z)$

Now, only the clause 2 can be used for merge. The clauses 1 and 3 will suspend until a new binding is computed on y.

2.2. Guarded Horn Clauses.

The input-constraints of GHC require that the variables of the atom in the goal do not get instantiated during the derivation step (neither by the unification with the head of a clause, nor by the evaluation of the guard). The only exception concerns the *unification atoms*. They are atoms of the form $t = t'$ and they can be solved by unifying t and t'. The computed substitution is the mgu of t and t'. These atoms are the only ones that can produce bindings on the variables of the goal. It turns out that, since the variables in predicates defined by program clauses cannot be instantiated by head unification, the unification atoms cannot be defined in GHC.

Example 2.2. The process *merge* and the processes p and q of example 2.1 can be defined in GHC in the following way:

1) $merge(cons(x,y), y', z') \leftarrow \mid z' = cons(x,z), merge(y, y', z)$.
2) $merge(y, cons(x,y'), z') \leftarrow \mid z' = cons(x,z), merge(y, y', z)$.
3) $merge(nil, y, z) \leftarrow \mid z = y$.
4) $merge(y, nil, z) \leftarrow \mid z = y$.
5) $p(y'_1) \leftarrow \mid y'_1 = cons(a,y_1), p(y_1)$.
6) $p(y'_1) \leftarrow \mid y'_1 = nil$.
7) $q(y'_2) \leftarrow \mid y'_2 = cons(b,y_2), q(y_2)$.
8) $q(y'_2) \leftarrow \mid y'_2 = nil$.

The initial goal corresponding to the one of example 2.1 is:

$\leftarrow p(y_3), q(y'_3), merge(y_3,y'_3,z_3)$.

Note that the clauses for merge could also be defined in the following way:

1') $merge(y'', y', z') \leftarrow y'' = cons(x,y) \mid z' = cons(x,z), merge(y, y', z)$.

2') $merge(y, y'', z') \leftarrow y'' = cons(x,y') \mid z' = cons(x,z), merge(y, y', z)$.
3') $merge(y', y, z) \leftarrow y' = nil \mid z = y$.
4') $merge(y, y', z) \leftarrow y' = nil \mid z = y$.

2.3. PARLOG.

For each predicate occurring in a PARLOG program there is an associated *declaration mode*. A declaration mode specifies, for each argument, whether it is an *input argument* (represented by the symbol ?) or an *output argument* (represented by the symbol ^). For instance, the declaration mode

$$p(?,?,\hat{})$$

specifies that the first two arguments of the predicate p are *input*, and that the third one is *output*. The input-constraints of PARLOG require that the variables in the input arguments of a process are not instantiated during the derivation step (neither by the head unification, nor by the guard evaluation).

Example 2.3. The process *merge* and the processes p and q of example 2.1. can be defined in PARLOG in the following way:

Declaration modes

$merge(?,?,\hat{})$
$p(\hat{})$
$q(\hat{})$

Program

1) $merge(cons(x,y), y', cons(x,z)) \leftarrow \mid merge(y, y', z)$.
2) $merge(y, cons(x,y'), cons(x,z)) \leftarrow \mid merge(y, y', z)$.
3) $merge(nil, y, y) \leftarrow \mid$.
4) $merge(y, nil, y) \leftarrow \mid$.
5) $p(cons(a,y_1)) \leftarrow \mid p(y_1)$.
6) $p(nil) \leftarrow \mid$.
7) $q(cons(b,y_2)) \leftarrow \mid q(y_2)$.
8) $q(nil) \leftarrow \mid$.

The initial goal corresponding to the one of example 2.1. is:

$\leftarrow p(y_3), q(y'_3), merge(y_3, y'_3, z_3)$.

3. A general theory for Read-Only unification.

In this section we define the extended notions of substitution and unification for dealing with the input constraints. We will use a uniform formalism based on the notion of read-only variables. This turns out to be very natural for expressing the synchronization mechanism of CP. The other two languages can be easily interpreted in this formalism by means of a simple translation.

We consider the two following countable sets:

- the set WV of the *writable variables*, with typical elements $x, y, z...$, and

- the set RV of the *read-only variables*, with typical elements $x?, y?, z?....$

We assume this sets to be isomorphic, and we assume the *read-only operator* $?: WV \rightarrow RV$ such that $?(x) = x?$ to be the isomorphism between WV and RV. The elements of the set $V = WV \cup RV$ will be denoted by $v, w....$

Let C be a finite set of *constructors symbols*. Each symbol is associated with an *arity*. The constants (arity 0) are denoted by $a,b,c...$, and the symbols with arity greater than 0 are denoted by $f, g, h....$ The set T, with typical elements $t, u...$, is the set of terms built on C and V. We extend the read-only operator to terms, i.e. we assume the existence of a function $?: T \rightarrow T$. For the moment, we do not define such a function, since we want to introduce a general framework to deal with different kinds of read-only constraints. Indeed, different theories for the read-only unification can be derived by considering different definitions of $?: T \rightarrow T$. We will show in the following that the difference between the synchronization mechanisms of CP, GHC, and PARLOG just relies on this function.

A *substitution* is a mapping $\vartheta: V \rightarrow T$ such that the set $D(\vartheta) = \{ v \mid \vartheta(v) \neq v \}$ (*domain* of ϑ) is finite. We will represent ϑ by the (finite) set $\{v/t \mid v \in Dom(\vartheta), \vartheta(v) = t\}$. The *application* of a substitution ϑ to a term t, $t\vartheta$, is obtained by simultaneously replacing every variable v occurring in t by $\vartheta(v)$. The *composition* $\vartheta\sigma$ of two substitutions ϑ and σ is defined as the composition of functions, i.e. $\vartheta\sigma(v) = (\vartheta(v))\sigma$ for every variable v. Two substitutions ϑ and σ are *equivalent* iff there exist a *renaming* [Apt88] γ such that $\vartheta\gamma = \sigma$. Let M be a set of sets of terms (or atoms). A substitution ϑ is a *unifier* for M iff

$$\forall S \in M \ \forall t,u \in S \ (t\vartheta = u\vartheta)$$

i.e., iff every set in M, under the application of ϑ, becomes a singleton. Given a set of sets M, a unifier ϑ is a *most general unifier* (*mgu*) iff for every unifier σ there exists a substitution γ such that $\vartheta\gamma = \sigma$. The set of the idempotent most general unifiers of M will be denoted by $mgu(M)$. There exist various algorithms for computing the elements of $mgu(M)$ (*unification algorithms*). A simple one, based on the Herbrand's original algorithm, is described in [Apt88]. A more efficient one has been defined by Martelli and Montanari [Martelli82]. Let ϑ be a substitution. The set of sets associated to ϑ, $S(\vartheta)$, is

$$S(\vartheta) = \{\{v, t\} \mid v/t \in \vartheta\}.$$

If Θ is a set of equivalent substitutions, we define

$$S(\Theta) = S(\vartheta)$$

where ϑ is any element of Θ.

We define now the notion of *read-only most general unifier*. Given a set of terms S, the *read-only correspondent* of S, $S?$, is defined by

$$S? = \{t? \mid t \in S\}.$$

Analogously, given a set of sets M, the *read-only correspondent* of M, $M?$, is defined by

$$M? = \{S? \mid S \in M\}.$$

The set of *read-only most general unifiers* of a set of sets M, $mgu_?(M)$, is defined as follows

$$mgu_?(M) = mgu(S(mgu(M)) \cup S(mgu(M))?).$$

We define now the *read-only unification result* and we discuss the case in which the unification yields a suspension. We assume the notion of *admissible* read-only mgu We will define this notion later.

Different definitions are possible, depending on the degree of atomicity that we want to model in the unification process. Given a set of sets M, the read-only unification result $R(M)$ can be either a substitution or one of the two special symbols *fail* and *suspend*, according to the following rules

$$R(M) = \begin{cases} \vartheta & \text{if } \exists \vartheta \in mgu_?(M) \text{ which is admissible} \\ fail & \text{if } mgu_?(M)=\emptyset \\ suspend & \text{otherwise} \end{cases}$$

The choice of ϑ, in this definition, is not relevant. In fact, it is well known that all the mgu's of a given set M are equivalent [Apt88]. However, it is in general necessary to compute more than one element of $mgu_?(M)$, before finding an admissible one (if any).

4. Read-only unification in CP, GHC and PARLOG.

In this section we show that the synchronization mechanisms of CP, GHC and PARLOG can be modeled as instances of our theory. In each case, we need essentially to define properly the function $?: T \to T$.

4.1. Concurrent Prolog.

In order to deal with the read-only unification in CP we define the function $?: T \to T$ as follows

Definition 4.1.

$$t? = \begin{cases} x? & \text{if } t = x \in W\,V \\ x & \text{if } t = x? \in R\,V \\ t & \text{otherwise} \end{cases}$$

We still have to define what we consider to be an admissible mgu. There are two possible "reasonable" definitions:

Definition 4.2. (Admissibility 1) $\vartheta \in mgu_?(M)$ is admissible$_1$ iff for any $x? \in Dom(\vartheta)$ we have $x \in Dom(\vartheta)$.

Definition 4.3. (Admissibility 2) $\vartheta \in mgu_?(M)$ is admissible$_2$ iff the restriction of ϑ to the writable variables (i.e. the substitution $\{x/\vartheta(x) \mid x \in Dom(\vartheta) \cap WV\}$) is a unifier for M.

We will call $R_1(M)$ and $R_2(M)$ the notions of read-only unification result, obtained by applying the first and the second notion of admissibility, respectively. Both these definitions are *order-independent*, in the sense that they don't depend on the order in which the elements of M are considered, when we apply the unification algorithm. This is quite important in order to preserve the declarativity of logic languages. The first definition of admissibility is "more parallel" than the second one. Intuitively, the first one allows a read-only variable $x?$ to get bound during the unification process by a binding generated on x.

Example 4.4. Consider the set (of sets) $M = \{\{f(x,x?), f(a,a)\}\}$. We have $mgu(M) = \{\vartheta\}$ where $\vartheta = \{x/a, x?/a\}$. Then $S(\vartheta) = S(\vartheta) \cup S(\vartheta)? = \{\{x,a\}, \{x?,a\}\}$ and $mgu_?(M) = mgu(S(\vartheta) \cup S(\vartheta)?) = \{x/a, x?/a\}$. With the first notion of admissibility we have $R_1(M) = \{x/a, x?/a\}$. With the second one, we have $R_2(M) = suspend$.

Example 4.5. Consider the set (of sets) $M = \{\{f(x,g(x?)), f(y,y)\}\}$. We have $mgu(M) = \{\vartheta\}$ where $\vartheta = \{x/g(x?), y/g(x?)\}$. Then $S(\vartheta) = \{\{x,g(x?)\}, \{y,g(x?)\}\}$ and $S(\vartheta)? = \{\{x?,g(x?)\}, \{y?,g(x?)\}\}$. Then $mgu_?(M) = mgu(S(\vartheta) \cup S(\vartheta)?) = \emptyset$. With both the notions of admissibility we have $R_1(M) = R_2(M) = fail$.

Example 4.6. Consider the set (of sets) $M = \{\{f(x,x?), f(g(y),g(y?))\}\}$. We have $mgu(M) = \{\vartheta\}$ where $\vartheta = \{x/g(y), x?/g(y?)\}$. Then $S(\vartheta) = \{\{x,g(y)\}, \{x?,g(y?)\}\}$ and $S(\vartheta)? = \{\{x?,g(y)\}, \{x?,g(y?)\}\}$. Then $mgu_?(M) = mgu(S(\vartheta) \cup S(\vartheta)?) = \{\vartheta_1, \vartheta_2\}$ where $\vartheta_1 = \{x/g(y?), x?/g(y?), y/y?\}$, and $\vartheta_2 = \{x/g(y), x?/g(y), y?/y\}$. We note that ϑ_1 is admissible$_1$, not admissible$_2$, and ϑ_1 is neither admissible$_1$, nor admissible$_2$. Then, with the first notion of admissibility we have $R_1(M) = \{x/g(y?), x?/g(y?), y/y?\}$. With the second one, we have $R_2(M) = suspend$.

The definition of $mgu_?$ allows to define the operational semantics of CP without considering a definition "ad hoc" for the application of a substitution. We use a transition system $(S, FS, \to)$, where

- S is the set of states, representing the configurations a computation proceeds through. A state is a pair $\langle G, \vartheta \rangle$, where G is either a goal or one of the special symbols *suspend* or *fail*, and ϑ is a (annotated) substitution
- FS is the set of final states $\langle G, \vartheta \rangle$ such that G is the empty goal (*success*), or *suspend*, or *fail*.
- $\to$ is the transition relation, defined by the following rules

1) $\langle \leftarrow \ldots A \ldots, \vartheta \rangle \to \langle \leftarrow (..B_1,\ldots B_n \ldots)\psi, \vartheta\psi \rangle$
if there exists $H \leftarrow G_1,\ldots G_m \,|.\, B_1,\ldots B_n$ (renamed) such that $R(\{H,A\}) = \sigma$ and $\langle \leftarrow (G_1,\ldots G_m)\sigma, \sigma \rangle \to^* \langle \square, \psi \rangle$

2) $\langle \leftarrow \ldots A \ldots, \vartheta \rangle \to \langle fail, \vartheta \rangle$
if for all the clauses $H \leftarrow G_1,\ldots G_m \,|.\, B_1,\ldots B_n$ (renamed) we have either $R(\{H,A\}) = fail$ or $R(\{H,A\}) = \sigma$ and $\langle \leftarrow (G_1,\ldots G_m)\sigma, \sigma \rangle \to^* \langle fail, \psi \rangle$

3) $\langle \leftarrow A_1,\ldots A_n, \vartheta \rangle \to \langle suspend, \vartheta \rangle$
if for all the A_i's, for all the clauses $H \leftarrow G_1,\ldots G_m \,|.\, B_1,\ldots B_n$ (renamed) we have
i) either $R(\{H,A\}) = fail$, or
ii) $R(\{H,A\}) = suspend$ or $R(\{H,A\}) = \sigma$ and $\langle \leftarrow (G_1,\ldots G_m)\sigma, \sigma \rangle \to^* \langle suspend, \psi \rangle$ and there exists at least one clause that satisfies the condition **(ii)**.

Example 4.7. Consider the CP program of example 2.1. Consider the initial goal

$\leftarrow p(y_3), q(y'_3), merge(y_3?, y'_3?, z_3).$

The process *merge* is forced to wait until some bindings are produced on the variables y_3 or y'_3. Indeed, if we consider the clause 1 we have

$$mgu_?(\{\{merge(cons(x,y), y', cons(x,z)),\ \ merge(y_3?, y_3?, z_3)\}\}) =$$
$$\{\{y_3?/cons(x,y),\ y'/y'_3?,\ z_3/cons(x,z),\ y'?/y'_3?,\ z_3?/cons(x,z)\},...\}$$

and we have that both R_1 and R_2 *suspend*. We get the same result also considering all the other clauses for *merge*.

Assume now that p performs a step by using the clause 5. Then we have

$$mgu_?(\{\{p(cons(a,y_1)),\ p(y_3)\}\}) =$$
$$\{\{y_3/cons(a,y_1),\ y_3?/cons(a,y_1)\},...\}$$

and we have $R_1(\{\{p(cons(a,y_1)),\ p(y_3)\}\}) = R_2\ \{\{p(cons(a,y_1)),\ p(y_3)\}\} = \{y_3/cons(a,y_1),\ y_3?/cons(a,y_1)\}$. Then the goal becomes

$\leftarrow p(y_1), q(y'_3), merge(cons(a,y_1),y_3?,z_3)$

4.2. Guarded Horn Clauses and PARLOG.

In order to deal with the read-only unification in GHC and PARLOG we define the function $?: T \rightarrow T$ as follows

Definition 4.8.

$$t? = \begin{cases} x? & \text{if } t = x \in W\ V \\ x? & \text{if } t = x? \in R\ V \\ f(t_1?,...t_n?) & \text{if } t = f(t_1,...t_n) \end{cases}$$

It turns out that, with this definition of the function $?: T \rightarrow T$, we can define $mgu_?(M)$ in a easier way. This is shown by the following theorem.

Theorem 4.9. If $?: T \rightarrow T$ is defined as in definition 4.8, then $mgu_?(M) = mgu(M \cup M?)$ holds.

Proof. We show first that M and $S(mgu(M))$ have exactly the same unifiers, and then the same mgu's. Assume $\vartheta \in mgu(M)$. Let ϑ' be a unifier for M. Then $\vartheta' = \vartheta\gamma$, for an appropriate γ. Let $\vartheta(x) = t$. Since ϑ is idempotent, we have $x\vartheta' = x\vartheta\gamma = t\gamma = t\vartheta\gamma = t\vartheta'$, i.e. ϑ' is a unifier for ϑ. Assume now ϑ' to be a unifier for $S(mgu(M))$. Then for all x, $x\vartheta' = x\vartheta\vartheta'$ holds. Therefore $\vartheta' = \vartheta\vartheta'$ and then $M\vartheta' = M\vartheta\vartheta'$, i.e. ϑ' is a unifier for M.

Analogously, $M?$ and $S(mgu(M?))$ have the same mgu's. Moreover, the function $?: T \rightarrow T$ of definition 4.8 propagates the annotations "?" to the variables inside terms. Therefore it is easy to see that $S(mgu(M?)) = S(mgu(M))?$ holds. Then we have $u_?(M) = mgu(S(mgu(M)) \cup S(mgu(M))?) = mgu(M \cup M?)$. □

Example 4.10. Consider the same M as in example 4.6. As already shown, $mgu(M) = \{\vartheta\}$, where $\vartheta = \{x/g(y),\ x?/g(y?)\}$, and $S(\vartheta) = \{\{x,g(y)\},\ \{x?,g(y?)\}\}$. The first difference is in the definition of $S(\vartheta)?$, indeed we have $S(\vartheta)? = \{\{x?,g(y?)\}\}$. Then $mgu_?(M) = mgu(S(\vartheta) \cup S(\vartheta)?) = \{\vartheta'\}$ where $\vartheta' = \{x/g(y),\ x?/g(y?)\}$. With the first notion of admissibility we have $R_1(M) = \{x/g(y),\ x?/g(y?)\}$. With the second one, we have $R_2(M) = suspend$.

We show now how to deal with GHC and PARLOG programs by using our formalism. The idea is to translate these programs into CP-like programs, and then to apply the definition of $mgu_?$ given for GHC and PARLOG.

4.2.1. Guarded Horn Clauses.

Given a GHC clause $A \leftarrow G_1,...,G_m \mid B_1,...,B_n$. we translate it into a CP clause $A \leftarrow G'_1,...,G'_m \mid B'_1,...,B'_n$.

where

$$G'_i\ [B'_i] = \begin{cases} G_i\ [B_i] & \text{if } G_i\ [B_i] \text{ is a unification atom } t = t' \\ G_i?\ [B_i?] & \text{otherwise} \end{cases}$$

and $?: T \rightarrow T$ is defined as in definition 4.8. Analogously, given a GHC goal $\leftarrow B_1,...,B_n$. we translate it into a CP goal $\leftarrow B'_1,...,B'_n$. where the B'_i's are defined as before.

Example 4.11. Consider the GHC program of example 2.2. Its translation is

1) $merge(cons(x,y), y', z') \leftarrow \mid z' = cons(x, z), merge(y?, y'?, z?).$
2) $merge(y, cons(x,y'), z') \leftarrow \mid z' = cons(x, z), merge(y?, y'?, z?).$
3) $merge(nil, y, z) \leftarrow \mid z = y.$
4) $merge(y, nil, z) \leftarrow \mid z = y.$
5) $p(y'_1) \leftarrow \mid y'_1 = cons(a,y_1),\ p(y_1?).$
6) $p(y'_1) \leftarrow \mid y'_1 = nil.$
7) $q(y'_2) \leftarrow \mid y'_2 = cons(b,y_2),\ q(y_2?).$
8) $q(y'_2) \leftarrow \mid y'_2 = nil.$

The initial goal is translated into

$\leftarrow p(y_3?), q(y'_3?), merge(y_3?,y'_3?,z_3?).$

4.2.2. PARLOG.

Given a PARLOG clause $A \leftarrow G_1,...,G_m \mid B_1,...,B_n$, we translate it into a CP clause $A \leftarrow G'_1,...,G'_m \mid B'_1,...,B'_n$.

where $G'_i\ [B'_i] = p(t'_1,...,t'_k)$, if $G_i\ [B_i] = p(t_1,...,t_k)$, and

$$t'_j = \begin{cases} t_j? & \text{if the } j\text{-th argument of } p \text{ is declared as an input argument} \\ t_j & \text{if the } j\text{-th argument of } p \text{ is declared as an output argument} \end{cases}$$

Analogously, given a PARLOG goal $\leftarrow B_1,...,B_n$. we translate it into a CP goal $\leftarrow B'_1,...,B'_n$.where the B'_i's are defined as before.

Example 4.11. Consider the PARLOG program of example 2.2. Its translation is

1) $merge(cons(x,y), y', cons(x,z)) \leftarrow \mid merge(y?, y'?, z)$.
2) $merge(y,cons(x,y'), cons(x,z)) \leftarrow \mid merge(y?, y'?, z)$.
3) $merge(nil, y, y) \leftarrow \mid$.
4) $merge(y, nil, y) \leftarrow \mid$.
5) $p(cons(a,y_1)) \leftarrow \mid p(y_1)$.
6) $p(nil) \leftarrow \mid$.
7) $q(cons(b,y_2)) \leftarrow \mid q(y_2)$.
8) $q(nil) \leftarrow \mid$.

The initial goal is translated into

$\leftarrow p(y_3), q(y'_3), merge(y_3?, y'_3?, z_3)$.

The operational semantics of a PARLOG or GHC goal G, w.r.t. a program P, can be described via the operational semantics of the translated goal G' in the translated program P'. The transition system is the same as the one defined in section 4.1., apart from the following

- the function R, that is defined in a way different from the one of CP, and
- the rule **(1)** must be redefined in the following way:

$$< \leftarrow ...A...,\vartheta> \rightarrow < \leftarrow ((.....)\psi, (B_1,...B_n)\ disannotate(\psi)),\vartheta\psi> \text{ if there exists....}$$

instead of

$$< \leftarrow ...A...,\vartheta> \rightarrow < \leftarrow (..B_1,...B_n...)\psi, \vartheta\psi> \text{ if there exists....}$$

where $disannotate(\psi)$ eliminates all the annotations from the terms of the domain and the codomain of ψ. This is necessary because CP and PARLOG have inheritance rules different from the ones of CP. In CP the read-only annotation is inherited both by the guard and the body. In the other two languages the input constraints on the variables of the goal are active during the evaluation of the guard, but are discharged after the commitment.

Example 4.11.

a) Consider the following CP program

1) $p(x) \leftarrow q(x) \mid r(x)$.
2) $q(x) \leftarrow \mid$.
3) $r(a) \leftarrow \mid$.

and the goal

$\leftarrow p(y?)$

we have $R(\{p(y?)\ ,\ p(x)\}) = \{x/y?,\ x?/y?\}$. Then, the guard $\leftarrow q(y?)$ is evaluated successfully and the goal reduces to the new goal

$\leftarrow r(y?)$

that yields a failure. This is perfectly in accordance with the standard CP semantics.

b) Consider now the following GHC program

1) $p(x) \leftarrow q(x) \mid x = a$.
2) $q(x) \leftarrow \mid$.

that is equal to its translation, and the goal

$\leftarrow p(y)$

translated into

$\leftarrow p(y?)$

as before, we have $R(\{p(y?)\ ,\ p(x)\}) = \{x/y?,\ x?/y?\}$. Then, the guard $\leftarrow q(y?)$ is evaluated successfully and we get the new goal

$\leftarrow (x = a)disannotate(\{x/y?,\ x?/y?\})$

i.e.

$\leftarrow y = a$

that is refutable. This is in accordance with the standard GHC semantics of the goal $\leftarrow p(y)$.

5. Related works.

The declarative definition of the read-only unification in CP has been first investigated by Shapiro [Shapiro87] and successively by Szoke [Szoke88]. Their approach is quite similar to the one defined in this paper for CP. They define $mgu_?(M)$ in the following way:

$$mgu_?(M) = mgu(M) \cup mgu(M)?.$$

The main problem with this definition is that the substitution computed in this way may be not a read-only unifier, and, in general, is not idempotent. Consider the following examples:

Example 5.1. Let M be the set $\{\{f(x,f(x?)), f(y,y)\}\}$ of example 4.5. With the definition of Shapiro and Szoke we have $mgu_?(M) = \{\{x/f(x?), x?/f(x?), y/f(x?), y?/f(x?)\}\}$. This is not a unifier for M (and it is not idempotent). Indeed, the M does not have any read-only unifier. This is dealt correctly by our definition, in which we get $mgu_?(M) = \emptyset$.

Example 5.2. Let M be the set $\{\{f(x,f(x?)), f(a,z)\}\}$. With the definition above we have $mgu_?(M) = \{\{x/a, x?/a, z/f(x?), z?/f(x?)\}\}$. This is not a unifier for M (and it is not idempotent). By our definition , we get $mgu_?(M) = \{\{x/a, x?/a, z/f(a), z?/f(a)\}\}$, that unifies M.

An other difference with these works relies in the notion of *result*: Their notion corresponds to the one that we call R_2 , and they do not investigate the one that models the parallel read-only unification, namely R_1.

A different definition of constrained unification, in GHC and in PARLOG, has been investigated in [Palamidessi88, deBoer89]. The definition given there is equivalent to the one given in this paper (for GHC and PARLOG), but much more complicated, and it is very difficult to see whether or not the

properties of the standard theory of unification are preserved. This is not the case here, since we extend naturally the standard definition of substitutions, unification etc. Moreover, in that case is still necessary to define an algorithm "ad hoc" to compute the $mgu_?$, whilst here we define $mgu_?$ by using the standard notion of *mgu*.

Acknowledgements. I would like to thank Giorgio Levi, Maurizio Gabbrielli and Paolo Ciancarini for many stimulating and helpfull discussions about the subject of this paper.

6. References.

[Apt88] K.R. Apt. Introduction to Logic Programming. (revised version) Report CS-R8826, CWI, Amsterdam (1988). To appear on Handbook of Theoretical Computer Science, van Leeuwen, J. Managing Editor, North Holland.

[deBoer88] F.S. de Boer, J.N. Kok, C. Palamidessi and J. Rutten. Semantics models for PARLOG. Proc. of the Sixth International Conference on Logic Programming, Lisbon, 19 - 23 June (1989).

[Ciancarini89] P. Ciancarini, C. Palamidessi. A Hierarchy of Uniform Interpreters for Stream-Parallel Logic Languages. Draft. Dip. di Informatica, Pisa, 1989.

[Clark85] K.L. Clark and S. Gregory. Notes on the implementation of PARLOG, *Journal of Logic Programming, 2(1)*, (1985), 17-42.

[Clark86] K.L. Clark and S. Gregory. PARLOG: Parallel programming in logic. *ACM Trans. on Progr. Lang. and Syst. 8* (1986), 1-49.

[vanEmden76] M.H. van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. *J. ACM 23* (1976), 733-742.

[Falaschi89] M. Falaschi, M. Gabbrielli, G. Levi, M. Murakami. Nested Guarded Horn Clauses: a language provided with a complete set of Unfolding Rules. To appear in *Proc. of LPC '89*. Tokyo, (1989).

[Gabbrielli89] M. Gabbrielli, G. Levi. A semantic reconstruction of concurrent constraint logic programming. Draft. Dip. di Informatica, Pisa (1989).

[Gregory87] S. Gregory. Parallel Logic Programming in PARLOG, International Series in Logic Programming, Addison-Welsey Pub. Comp.(1987).

[Jaffar87] J. Jaffar and J-L. Lassez. Constraint Logic Programming. Proc. SIGACT-SIGPLAN, ACM, (1987), 111-119.

[Levi85] G. Levi and C. Palamidessi. The declarative semantics of logical read-only variables. *Proc. 1985 IEEE Symposium on Logic Programming*, IEEE Comp. Society Press, (1985), 128-137.

[Lloyd87] J.W. Lloyd. Foundations of logic programming. Second Edition, Springer-Verlag, (1987).

[Maher87] M. Maher. Logic Semantics for a class of committed-choice programs. 4th Int. Conf. on Logic Programming, MIT press (1987).

[Martelli82] A. Martelli and U. Montanari. An efficient Unification Algorithm. *ACM Trans. on Progr. Lang. and Systems, vol. 4, No 2*, (1982), 258-282.

[Palamidessi88] C. Palamidessi. A Fixpoint Semantics for Guarded Horn Clauses. Report CS-R8833, CWI, Amsterdam.

[Saraswat89] V. A. Saraswat. Concurrent Constraint Programming Languages. PhD. thesis. Carnegie-Mellon University, (1989).

[Shapiro83] E.Y. Shapiro. A subset of Concurrent Prolog and its interpreter. ICOT Technical Report TR-003 (1983).

[Shapiro86] E.Y. Shapiro. Concurrent Prolog: A progress report. In Fundamentals of Artificial Intelligence, W. Bibel and Ph. Jorrand, Eds. LNCS 232. Springer-Verlag, (1986), 277-313.

[Shapiro87] E.Y. Shapiro. The Family of Concurrent Logic Languages. Technical Report, Weizmann Institute of Science (1988).

[Shapiro88] E.Y. Shapiro. Concurrent Prolog: A progress report. In Concurrent Prolog, Collected Papers, E.Y. Shapiro, Eds. MTI press (1987), Chapt. 5, 157-187 .

[Szoke88] D. Szoke. Distributed Implementation of Flat Concurrent Prolog. Master's Thesis: Dept. of Computer Science, Weizmann Institute of Science (1988).

[Ueda85] K. Ueda. Guarded Horn clauses. ICOT Tech. Rep. TR-103 (1985).

[Ueda86] K. Ueda, Guarded Horn clauses. Doctoral Thesis, Information Engineering Course, Faculty of Engineering, University of Tokyo (1986).

La Semantica degli Aggiornamenti su Programmi Logici di Horn‡

Luigi Palopoli

CRAI, 87036 Rende - S.Stefano (CS)

SOMMARIO. L'assenza di una semantica dichiarativa ben definita degli operatori di aggiornamento del Prolog, *assert* e *retract*, comporta che la loro caratterizzazione operazionale sia spesso dipendente dall'implementazione ed, in taluni casi, controintuitiva. Pertanto in questo lavoro si propone una semantica dichiarativa per le operazioni di aggiornamento nella programmazione logica. In particolare vengono considerate operazioni di aggiornamento che, espresse in forma di goal, specifichino inserimenti o cancellazioni di atomi in un programma logico di Horn.

1. Introduzione.

I linguaggi logici, ed in particolare il Prolog [1] si sono ampiamente affermati in questi ultimi anni come *buoni* strumenti di programmazione, particolarmente in riferimento ad applicazioni software *avanzate* (IA, basi di dati deduttive, etc.) ed alla prototipizzazione rapida di sistemi. Tale successo e` stato probabilmente motivato dall'elevato grado di dichiarativita` che caratterizza i linguaggi logici: il programmatore specifica la costruzione di un certo oggetto descrivendone solo le caratteristiche, e non il modo in cui questo dovra` essere costruito. In tal senso in letteratura ci si e` spesso riferiti ai programmi logici come *specifiche eseguibili*.

Da un punto di vista formale la semantica di un programma logico (senza negazione) viene descritta dal suo minimo modello di Herbrand. Tale modello viene caratterizzato come intersezione di tutti i modelli del programma logico ed inoltre coincide con il minimo punto fisso di una trasformazione naturale associata al programma sul reticolo delle sue interpretazioni [2].

Al fine di specificare, anche dinamicamente, modifiche della base dei fatti e delle regole di un programma, Prolog fornisce al programmatore degli appositi metapredicati - *assert* per le inserzioni e *retract* per le cancellazioni -, che possono essere sia specificati come goal sia utilizzati nei corpi delle clausole. Per la loro stessa natura - descrivono, infatti, "transizioni di stato" -, la semantica di tali operatori e` difficilmente descrivibile in maniera formale facendo appello agli strumenti matematici che vengono usati per descrivere la semantica dei programmi logici (modelli di Herbrand, punti fissi, etc.). Di conseguenza, le funzionalita` specifiche degli operatori di *assert* e *retract* possono variare da implementazione ad implementazione. Inoltre, i risultati prodotti dall'esecuzione di tali operatori sono, talvolta, abbastanza controintuitivi.

‡ Ricerca parzialmente finanziata dal C.N.R., nell'ambito del PFI-SP5, obbiettivo Logidata+.

Lo scopo di questo lavoro e` quello di proporre una semantica dichiarativa formale per le operazioni di aggiornamento dei programmi logici. La semantica proposta cerca di rimanere il piu` vicino possibile al significato intuitivo delle operazioni trattate. Piu` in particolare, la trattazione sara` limitata ad una particolare classe di aggiornamenti, che si ritiene essere sufficientemente significativa. Tale classe e` costituita da tutti quegli aggiornamenti che specifichino inserzioni/cancellazioni di fatti (non necessariamente ground) in/da un programma logico, e che siano espressi in forma di goal. In altri termini non saranno considerati: (1) inserzioni/cancellazioni di regole, e (2) aggiornamenti specificati nei corpi delle clausole.

Il problema che viene affrontato in questo lavoro ha molte attinenze con quello della definizione della semantica degli aggiornamenti delle relazioni intensionali nelle basi di dati deduttive [3,4,5,6,7]. Ma differenza fondamentale tra i due problemi e` che nel secondo l'enfasi viene generalmente posta sulla corretta "traduzione" di un aggiornamento di una relazione intensionale, specificata attraverso un insieme di clausole, in un insieme di aggiornamenti delle relazioni di base. Questo approccio non sembra giustificabile nel caso di programmi logici, dal momento che presupporrebbe l'adozione di una distinzione netta tra conoscenza aggiornabile (le relazioni di base) e conoscenza non aggiornabile (le relazioni intensionali), distinzione che di fatto risulta controintuitiva in molti domini applicativi propri della programmazione logica, prima di tutti l'intelligenza artificiale.

Nel seguito verra` usata una sintassi diversa da quella del Prolog per la specificazione di operazioni di inserimento o cancellazione, in quanto si vuole enfatizzare una differenza fondamentale esistente tra la semantica degli operatori di aggiornamento, cosi` come viene specificata in questo lavoro, e quella che viene normalmente associata ai corrispondenti operatori del Prolog: cio` che viene aggiornato nel nostro caso e` l'insieme delle conseguenze logiche di un programma e non, come nel caso del Prolog, solo i fatti che fanno parte del programma stesso. Tale differenziazione verra` a piu` riprese sottolineata nella presentazione, attraverso l'uso di esempi.

Il lavoro e` organizzato come segue. Nel paragrafo seguente verranno richiamati alcuni concetti fondamentali. Nel terzo paragrafo la semantica degli aggiornamenti nei programmi logici verra` discussa in maniera informale, facendo costante riferimento alla semantica corrispondente di Prolog - come interprete di riferimento sara` utilizzato *C-Prolog 1.5* -. Nel quarto paragrafo sara` presentata la semantica formale degli aggiornamenti nei programmi logici, e ne verranno illustrate alcune proprieta`. Infine il quinto paragrafo conterra` alcune considerazioni conclusive.

2. Concetti di base.

In questo paragrafo vengono brevemente richiamati alcuni concetti fondamentali che riguardano la programmazione logica. Per una trattazione completa si veda [2]. Un programma logico (di Horn) e` un insieme di fatti e regole (di Horn). Una regola e` costituita da due parti: un letterale positivo (testa, o conseguente), scritto sulla sinistra del segno di implicazione logica $\leftarrow$, e una congiunzione di letterali (corpo), scritta sulla destra del segno di implicazione logica. Per esempio,

$$p(X) \leftarrow q(f(X)), \neg r(X)$$

e` una regola in cui $p(X)$ e` la testa, $q(f(X))$ e $\neg r(X)$ sono letterali. Una regola di Horn e` una regola nel cui corpo non compaiono letterali negativi. Un fatto e` una regola con corpo vuoto. Un goal e` una regola con testa vuota. Come in Prolog, le variabili sono stringhe alfanumeriche che iniziano con una lettera maiusola. Per ogni simbolo di predicato $p$, $\alpha(p)$ denota l'arita` di p.

L'insieme dei simboli di predicato di un programma logico P sara` indicato con $S_P$. L'universo di Herbrand $U_P$ di $P$ e` l'insieme di tutti i termini che possono essere costruiti utilizzando le costanti ed i funtori di P. Nel caso che nessuna costante appaia in P, si suppone l'esistenza di una costante arbitraria. La base di Herbrand B_P di P e` l'insieme di tutti gli atomi ground (cioe`, senza variabili) che possono essere costruiti utilizzando i simboli di predicato di S_P ed i termini di U_P. Una istanza ground di una regola di P viene costruita da una regola di P sostituendo consistentemente le sue variabili con elementi di U_P. Un'interpretazione (di Herbrand) di P e` un qualsiasi sottoinsieme di $B_P$. Una sostituzione $\theta$ e` un insieme di coppie $\{X_1/t_1,...,X_n/t_n\}$ tali che $X_1,...,X_n$ sono variabili tra loro distinte, $t_1,...,t_n$ sono termini ed ogni t_i e` distinto dalla corrispondente variabile $X_i$, $1 \le i \le n$. Una sostituzione e` ground se i termini t_i non contengono variabili, $1 \le i \le n$.

Un atomo ground $p(a) \in B_P$ e` vero rispetto ad un'interpretazione $I \subseteq B_P$ se $p(a) \in I$. Un letterale negativo $\neg p(a)$ e` vero rispetto ad I se il suo opposto $p(a)$ non e` vero rispetto ad $I$. Una regola $\rho \in P$ e` vera rispetto ad I se per ogni sostituzione θ, la testa di $\rho\theta$ appartiene ad I ogni qual volta il corpo di $\rho\theta$ e` vero rispetto ad $I$. $I$ e` un modello di P se tutte le regole e tutti i fatti di P sono veri rispetto ad I. Le proprieta` seguenti sono valide per ogni programma logico di Horn P [2]:

P1. (Proprieta` di intersezione tra modelli). Siano $M_1$ ed $M_2$ due modelli di $P$. Allora $M_1 \cap M_2$ e` pure un modello di P.

P2. Sia Γ_P l'insieme di tutti i modelli di P. Allora $LM_P = \bigcap_{M \in \Gamma_P} M$ e` il minimo modello di P.

Da qui in avanti, ci si riferira` solo a programmi logici di Horn e la presenza di letterali negativi sara` ammessa solo nei goal.

Un aggiornamento e` una richiesta di un insieme di inserimenti o di cancellazioni di atomi (non necessariamente ground) in un programma logico. Non sono ammessi inserimenti/cancellazioni di letterali negativi. Usaremo la sintassi $mta(p(t))$ ($mtr(p(t))$), per indicare una inserzione (cancellazione) in un programma logico di un fatto $p(t)$. I simboli *mta* e *mtr* non corrispondono a simboli di predicato (ne`, tantomeno, a funtori) e, conseguentemente, non verra` per essi definita nessuna nozione di verita`. Inserimenti e cancellazioni di atomi in un programma logico vengono specificati come goal posti al programma stesso. Per esempio il goal G:

$$\leftarrow ?\ \leftarrow p(b), mta(p(a))$$

specificato su un programm P richiede che il "nuovo stato" di P, ottenuto da quello "corrente" mediante l'esecuzione di G, sia tale che $p(a)$ sia deducibile dal "nuovo stato", se l'atomo $p(b)$ e` deducibile dallo "stato corrente", il "nuovo stato" sia identico a quello "corrente", altrimenti. Piu` formalmente un aggiornamento U (che andra` posto come goal ad un programma logico) ha la seguente forma:

$$U = L_1, u_1, ..., L_{n-1}, u_{n-1}, L_n$$

dove u_j e` $mtr(p(t))$ oppure $mta(p(t))$ per qualche atomo $p(t)$, ed $L_i$ e` una congiunzione di letterali (eventualmente vuota), $1 \le j \le n-1$, $1 \le i \le n$.

Nel prossimo paragrafo sara` introdotta informalmente la semantica degli aggiornamenti su programmi logici di Horn. Al fine di semplificare la presentazione, si fara` riferimento inizialmente ad aggiornamenti "semplici", e poi via via piu` complessi.

3. Un'introduzione informale.

Si consideri un programma logico P (di Horn) e si assuma che nessun aggiornamento sia stato eseguito su di esso. Come gia` sottolineato nel paragrafo introduttivo, la semantica di $P$ e` caratterizzata dal suo minimo modello di Herbrand LM_P. Si noti, infatti, che P ammette modello minimo, per la proposizione P2. Si consideri ora un aggiornamento u su P che agisca su un atomo $p(a)$, e di forma molto semplice - che consista, cioe`, di una singola occorrenza di *mtr* o di *mta* -. La semantica dichiarativa che verra` associata all'esecuzione di u su P sara` caratterizzata da una interpretazione $I_{u,P}$ di $P$ ottenuta quale variazione minima di $LM_P$ tale che $u$ e` *realizzato in* $I_{u,P}$, cioe`, $p(a)$ e` vero rispetto a $I_{u,P}$, se $u = mta(p(a))$, $p(a)$ e` falso in $I_{u,P}$, se $u = mtr(p(a))$. Si noti che non viene richiesto che $I_{u,P}$ sia un modello per P.

Esempio 1. Si consideri il seguente programma logico P:

$q(X) \leftarrow p(X)$.

$p(b)$.

$r(a)$.

E' facile verificare che $LM_P = \{p(b), q(b), r(a)\}$. Si consideri l'aggiornamento $u = mtr(q(b))$. Dunque si ottiene $I_{u,P} = \{p(b), r(a)\}$. Si noti che $I_{u,P}$ non e` un modello per P, e tuttavia esso corrisponde intuitivamente alla *realizzazione minima* di u rispetto a P. □

E' interessante considerare, a questo punto, la versione Prolog del programma P e dell'aggiornamento u dell'esempio precedente. Sia, dunque, P' il seguente programma Prolog:

$q(X)$:– $p(X)$.

$p(b)$.

$r(a)$.

Si consideri l'aggiornamento $u' = retract(q(b))$. L'interprete Prolog esegue u' su P' cercando tra i fatti di P' quelli che sono unificabili con $q(b)$, argomento del *retract*. Questi fatti vengono rimossi da P', e le corrispondenti unificazioni vengono generate. Si noti che la ricerca viene limitata ai soli fatti del programma logico in questione, e non estesa alle sue conseguenze logiche. Di conseguenza, nel caso in esempio, l'esecuzione dell'aggiornamento u' non provoca alcun cambiamento nel programma P', e la semantica complessiva dell'aggiornamento risulta $I_{u',P'} = LM_{P'}$.

Il precedente esempio mostra molto chiaramente quanto la semantica dell'operazione *retract* in Prolog dipenda dal modo in cui il programma a cui essa viene applicata e` scritto, piu` che dalla sua semantica. Si consideri, a questo proposito, il seguente programma Prolog P'':

$q(b)$.

$p(b)$.

$r(a)$.

E' chiaro che P'' e` semanticamente equivalente a P', essendo $LM_{P'} = LM_{P''}$. Di conseguenza sarebbe lecito aspettarsi che l'esecuzione di u' su P'' abbia lo stesso risultato ottenuto dall'applicazione di u' a P'. Invece, a causa della semantica operazionale del *retract* in Prolog, otteniamo:

$I_{u',P''} = \{p(b), r(a)\} \neq I_{u',P'}$

In altri termini, le operazioni di aggiornamento in Prolog sono fortemente "legate alla sintassi" dei programmi su cui tali aggiornamenti vengono eseguiti. Viceversa, la proposta presentata in questo lavoro prevede una semantica degli aggiornamenti "legata al modello" del programma logico in questione.

Consideriamo ora una forma piu` complessa di aggiornamento, in cui l'operazione di aggiornamento vera e propria (che anche in questo caso riguarda un atomo ground) e` accompagnata da una precondizione e da una postcondizione alla sua esecuzione. Sia, dunque, $U = L_1, u, L_2$, dove L_i, i=1, 2, e` una congiunzione di letterali (non necessariamente ground), ed $u$ e` un'operazione di aggiornamento ground semplice (cioe`, $u = mtr(p(a))$, oppure $u = mta(p(a))$, per qualche atomo ground $p(a)$). Intuitivamente la semantica dell'esecuzione di $U$ su un programma logico $P$ puo` essere descritta come segue: se esiste una sostituzione θ tale che $L_1\theta$ e` vero rispetto al minimo modello di $P$, ed $L_2\theta$ e` vero rispetto a $I_{u,P}$, allora la semantica complessiva $I_{U,P}$ dell'esecuzione di U su P e` data da $I_{U,P} = I_{u,P}$, altrimenti $I_{U,P} = LM_P$. In altri termini, $L_1$ fa da precondizione ed $L_2$ da postcondizione all'esecuzione dell'operazione di aggiornamento $u$. Si sottolinea che la semantica proposta supporta l'esecuzione di *transazioni* - cioe`, di aggiornamenti complessi condizionali da eseguire *atomicamente* - su un programma logico.

Da quanto detto in precedenza risultera` chiaro che, in generale, $I_{[L_1,u,L_2],P} \neq I_{[L_2,u,L_1],P}$. Pertanto, nel nostro approccio, permane una certa componente di operazionalita` (dipendenza dall'ordinamento dei sottogoal). Si consideri, tuttavia, che un certo grado di operazionalita` e` intrinseco nelle operazioni di aggiornamento, e non e`, se non forse artificiosamente, eliminabile. Un'esemplificazione di quanto appena affermato, e` data dagli aggiornamenti $U_1 = mta(p(a))$, $mtr(p(a))$ ed $U_2 = mtr(p(a))$, $mta(p(a))$, che differiscono tra loro solo nell'ordine in cui i sottogoal sono posti. Ora, per un qualunque programma logico P e` intuitivamente corretto che sia $I_{U_1,P} \neq I_{U_2,P}$.

Si noti che si sta richiedendo, per il momento, che negli aggiornamenti non compaiano variabili (il caso piu` generale dell'esecuzione di aggiornamenti in cui compaiano variabili verra` trattato nel seguito).

E' bene, inoltre, precisare che tale vincolo garantisce non solo che l'esecuzione di un aggiornamento u su un programma P avvenga solo se u e` ground, ma anche l'assenza di non-determinismo. Si supponga, infatti, di rilasciare tale vincolo e di imporre, viceversa, il seguente vincolo V:

(V) Se una variabile, ad esempio X, compare in un aggiornamento, allora essa compare anche in almeno un letterale positivo in una delle precondizioni a tale aggiornamento.

E' chiaro che V garantisce che qualsiasi aggiornamento u sia eseguito solo quando tutte le variabili eventualmente presenti in u siano state "rese bound" da qualche unificazione precedentemente generata. Sfortunatamente pero` esso non assicura la deterministicita` della semantica degli aggiornamenti, come viene esemplificato di seguito.

Esempio 2. Sia P il seguente programma logico:

$p(a, b)$.

$p(b, a)$.

Si consideri l'aggiornamento $U = p(X,Y)$, $mta(q(X))$, $mtr(q(Y))$. E' chiaro che V e` valido per U.

Si noti che $p(X, Y)$ unifica con entrambi i fatti di P, con le sostituzioni $\theta_1 = \{X/a, Y/b\}$ e $\theta_2 = \{X/b, Y/a\}$, rispettivamente. Ora, sembra essere abbastanza naturale che entrambi gli aggiornamenti $U\theta_1$ e $U\theta_2$ siano eseguiti su P. Purtroppo in questo caso la semantica complessiva dell'aggiornamento diventa dipendente dall'ordine in cui questi due aggiornamenti vengono eseguiti. Infatti,

$$I_{[U\theta_1],\,[U\theta_2],P} = \{p(a, b), p(b, a), q(b)\} \neq \{p(a, b), p(b, a), q(a)\} = I_{[U\theta_2],\,[U\theta_2],P}$$

Ne consegue che la semantica complessiva dell'esecuzione di U su P e` inerentemente non deterministica. □

In questo lavoro limiteremo la nostra attenzione ad aggiornamenti deterministici.

Consideriamo ancora una volta la semantica di Prolog, questa volta riguardo ad aggiornamenti del tipo descritto in precedenza, in cui, cioe`, siano presenti precondizioni o postcondizioni all'esecuzione. In questo caso, e diversamente che nel nostro approccio, lo stato globale di un programma Prolog viene cambiato dall'esecuzione di un'operazione di *assert* o *retract* anche se il goal complessivo di cui tale operazione fa parte fallisce. Cio` ha la spiacevole conseguenza che la presenza di postcondizioni, perde di significato. Ad esempio, un atomo che e` stato asserito rimane nella base dei fatti di un programma Prolog anche se l'operazione di assert viene "backtracked over".

Esempio 3. Si consideri il seguente programma logico P.

$p(a)$.

$r(b)$.

Sia P' il corrispondente programma Prolog ($P' = P$!). Si consideri l'aggiornamento $U = mta(p(b)), r(a)$. Dovrebbe risultare chiaro che, secondo il nostro approccio, la semantica dell'esecuzione di U su P e` caratterizzata dall'interpretazione $I_{U,P} = \{p(a), r(b)\} = LM_P$, poiche` $r(a)$ non e` vero rispetto a $I_{[mta(p(b))],P}$, e, di conseguenza, l'aggiornamento $U$ non ha complessivamente alcun effetto. Viceversa, l'aggiornamento $U' = assert(p(b)), r(a)$ eseguito su $P'$ induce l'interpretazione $I_{U',P'} = \{p(a), r(b), p(b)\}$, poiche` l'esecuzione di $assert(p(b))$ causa l'inserimento del fatto $p(b)$ in P', che non viene rimosso allorche` $r(a)$ fallisce. □

Piu` generale, un aggiornamento (con pre/post-condizioni) avra` la forma $U = L_1,u_1,...,u_{n-1},L_n$ dove u_j e` $mta(p(a))$ oppure $mtr(p(a))$, per qualche atomo ground $p(a)$, $1 \leq j \leq n-1$, ed $L_i$ e` una congiunzione di letterali (eventualmente vuota), $1 \leq i \leq n$. Il significato intuitivo che verra` associato all'esecuzione di $U$ e` il seguente: se esiste un sostituzione θ tale che, $L_j\theta$, $1 \leq j \leq n$, e` vero rispetto all'interpretazione $I_{[u_1,...,u_{j-1}],\,P}$, allora la semantica $I_{U,P}$ dell'esecuzione di $U$ su $P$ e` definita da $I_{U,P} = I_{[u_1,...,u_{n-1}],\,P}$, altrimenti $I_{U,P} = LM_P$. In altri termini ogni congiunzione L_j, $1 \leq j \leq n$, fa da precondizione all'esecuzione dell'aggiornamento u_i, $i = j \neq n$, e da postcondizione all'esecuzione dell'aggiornamento u_k, $k = j-1 \neq 1$. Il fallimento di una sola di tali condizioni L_j comporta il fallimento dell'intero goal, e, in questo caso, l'effetto complessivo dell'aggiornamento e` nullo.

Esempio 4. Si consideri il programma P seguente.

$p(X, Z) \leftarrow q(X, Y), p(Y, Z)$.

$p(X, Y) \leftarrow q(X, Y)$.

$q(a, b)$.

$q(b, c)$.

Si puo` notare che in p viene calcolata la chiusura transitiva di q. Si consideri ora l'aggiornamento:

$$U = mta(q(c, a)), p(X, X), mtr(q(a, b))$$

La semantica intuitiva associata all'esecuzione di U su P e` la seguente: se l'inserzione del fatto $q(c, a)$ induce l'esistenza di almeno una coppia $(\hat{a}, \hat{a})$, per qualche costante $\hat{a}$, allora il fatto $q(a, b)$, qualora presente, viene cancellato, altrimenti l'effetto complessivo dell'aggiornamento e` nullo. In questo caso si ottiene quindi:

$$I_{U,P} = \{q(b, c), q(c, a), p(b, c), p(c, a), p(b, a)\}$$ □

Rilasciamo ora l'ipotesi che il singolo comando di aggiornamento sia ground quando viene eseguito. Consideriamo, dunque, aggiornamenti che hanno in generale la forma seguente:

$$U = L_1,u_1,...,u_{n-1},L_n$$

dove gli L_j sono congiunzioni di letterali (non necessariamente ground ed, eventualmente, vuote), e gli u_i sono singole operazioni di aggiornamento (cioe`, $u_i$ e` $mta(p(t))$ oppure $mtr(p(t))$, per qualche atomo $p(t)$ non necessariamente ground), $1 \leq j \leq n$, $1 \leq i \leq n-1$. Al fine di evitare il non determinismo, imponiamo un vincolo sulla sintassi degli aggiornamenti. Sia $Var(u)$ l'insieme delle variabili che occorrono nell'operazione di aggiornamento u. Similmente, sia $Var(L)$ l'insieme delle variabili che compaiono nella congiunzione L. Allora ogni aggiornamento $U = L_1,u_1,...,u_{n-1},L_n$ deve soddisfare il seguente vincolo DV:

(DV) $(Var(L_1) \cup ... \cup Var(L_j)) \cap Var(u_j) = \varnothing$, per ogni j, $1 \leq j \leq n-1$.

Ad esempio l'aggiornamento $U = p(X), mta(q(Y))$ ricade nella classe appena definita. Si noti che la forma di aggiornamento trattata nella parte precedente di questo paragrafo e` un caso particolare di questa. Al fine di semplificare la presentazione, consideriamo inizialmente il solo problema della definizione del significato della presenza di variabili libere nelle operazioni di aggiornamento. Si consideri, dunque, un aggiornamento u della forma $mta(p(t))$ oppure della forma $mtr(p(t))$ per qualche atomo non ground $p(t)$. La semantica che verra` associata all'esecuzione di $u$ su un programma logico $P$ e` un'interpretazione $I_{u,P}$ che viene ottenuta come la minima variazione di LM_P che realizza *universalmente* u, cioe`, per ogni sostituzione ground $\theta$, $I_{u,P}$ realizza $u\theta$. In altri termini le variabili "libere" in un'operazione di aggiornamento vengono interpretate come se fossero quantificate universalmente. La ragione di questa scelta risiede nella ricerca di uniformita` con un "nucleo" semantico minimo che si puo` ritrovare nell'implementazione delle operazioni corrispondenti del Prolog. Un approccio differente consiste nel dare un'interpretazione esistenziale alle variabili presenti nelle operazioni di aggiornamento (si veda, ad esempio, [8]).

E' bene sottolineare che nel calcolo delle sostituzioni da applicare alle operazioni di aggiornamento in cui compaiono variabili bisogna riferirsi all'universo di Herbrand "corrente" del programma in esame, visto che questo varia, in generale, man mano che gli aggiornamenti vengono eseguiti.

Esempio 5. Si consideri il seguente programma P:

$p(a)$.

Sia $U = mta(p(b)),mta(r(X))$ un aggiornamento da eseguire su P. Il dominio delle sostituzioni applicabili ad $mta(r(X))$ e` $H_P \cup \{b\}$. Di conseguenza la semantica dell'esecuzione di $U$ su $P$ e` caratterizzata da $I_{U,P} = \{p(a), p(b), r(a), r(b)\}$. □

E' possibile che un aggiornamento venga solo *parzialmente* annullato da uno successivo, come mostra il seguente esempio.

Esempio 6. Si consideri il seguente programma logico P.

$p(a)$.

Sia $U = mta(p(f(X))), mtr(p(f(f(f(X)))))$. Si noti che l'operazione $mtr(p(f(f(f(X)))))$ maschera parzialmente il risultato dell'operazione precedente $mta(p(f(X)))$. In particolare si ha:

$I_{U,P} = \{p(a), p(f(a)), p(f(f(a)))\}$ □

L'esempio precedente puo` essere utilizzato per mostrare come, talvolta, i risultati prodotti dall'interprete Prolog nell'esecuzione di operazioni di aggiornamento non corrispondano a quanto, intuitivamente, ci si aspetterebbe. Sia, dunque, P' il programma Prolog corrispondente al programma P dell'Esempio 6. ($P' = P$!), e sia $U' = assert((p(f(X))), retract(p(f(f(f(X))))))$. Il risultato prodotto dall'esecuzione di U' su P' in Prolog e` caratterizzato da $I_{U',P'} = \{p(a)\}$. Infatti l'interprete Prolog inserisce innanzitutto il fatto $p(f(X))$. nella base dei fatti di P', e successivamente rimuove da questa tutti quei fatti che unificano con $p(f(f(f(X))))$, argomento dell'operazione di *retract*.

Nel prossimo paragrafo definiremo formalmente la semantica associata all'esecuzione delle operazioni di aggiornamento, e ne illustreremo le proprieta`.

4. La semantica degli aggiornamenti nei programmi logici.

Sia $U = L_1,u_1,...,u_{n-1},L_n$ un aggiornamento da eseguire su un programma logico P. Si assuma che DV sia valido per U. Lo scopo che si vuole raggiungere e` la definizione formale dell'interpretazione $I_{U,P}$. Come gia` sottolineato in precedenza, la semantica del programma nel suo stato "iniziale" e` caratterizzata dal suo minimo modello di Herbrand $LM_P$. E' stato inoltre sottolineato che $I_{U,P}$ non e` necessariamente un modello per P (si veda l'Esempio 1.). Questo non dovrebbe tuttavia sorprendere dal momento che un programma, visto come teoria logica, deve essere modificato in seguito all'esecuzione di un aggiornamento e cio` si riflette nel fatto che, in generale, $I_{U,P}$ non e` un modello per P. Vedremo infatti, in questo paragrafo, come sia possibile definire, a livello sintattico, una struttura, che chiameremo *programma logico esteso* (*p.l.e.*, per brevita`), che corrisponde all'esecuzione di $U$ su $P$ e che avra` $I_{U,P}$ come suo "modello minimo" - una opportuna nozione di modello di un p.l.e. sara` definita successivamente -. Viene, dunque, introdotta di seguito la nozione di p.l.e.

Un *programma logico esteso* ExP e` una struttura $<P, FS>$, dove $P$ e` un insieme di regole (cioe`, un programma logico privo di fatti), ed $FS$ e` un insieme di coppie della forma $<p, [<A_{p,1}, D_{p,1}>, <A_{p,2}, D_{p,2}>,...]>$, una coppia per ogni simbolo di predicato $p \in S_P$, tale che per

ogni coppia $<A_{p,i}, D_{p,i}>$, si ha:

1. $A_{p,i} \cup D_{p,i}$ e` un insieme di ennuple di termini arita` $\alpha(p)$,
2. $A_{p,i} \cap D_{p,i} = \emptyset$,
3. se $A_{p,i}$ ($D_{p,i}$) contiene un ennupla in cui compare un termine non ground, allora $D_{p,i} = \emptyset$ ($A_{p,i} = \emptyset$).

Sia $maxl(p)$ il numero di coppie contenute nella lista L_p associata al simbolo di predicato p (i.e., $<p, L_p> \in FS$.

Viene definita ora la nozione di verita` e la semantica dichiarativa dei p.l.e. Sia, dunque, $ExP = <P, FS>$ un p.l.e. Definiamo l'universo di Herbrand $U_{ExP}$ e la base di Herbrand $B_{ExP}$ di $ExP$ come segue. $U_{ExP}$ e` l'insieme di tutti i termini ground che possono essere costruiti usando le costanti ed i funtori in P ed in FS. B_{ExP} e` l'insieme di tutti gli atomi ground che possono essere costruiti usando i simboli di predicato di $P$ e di $FS$ ed i termini in $U_{ExP}$. Una *e-interpretazione* (di Herbrand) di $ExP$ e` un qualsiasi sottoinsieme di B_{ExP}.

Sia $ExP = <P, FS>$ un p.l.e. ed I un'e-interpretazione per ExP. Un atomo ground $p(a) \in B_{ExP}$ e` vero rispetto ad $I$ se $p(a) \in I$. Un letterale negativo ground $\neg p(a)$ e` vero rispetto ad I se il suo opposto $p(a)$ non e` vero rispetto ad $I$. Una congiunzione di letterali ground $C$ e` vera rispetto ad I se ogni letterale di C e` vero rispetto ad $I$. Una istanza ground $\gamma$ di una regola di $P$ e` vera rispetto ad I se la testa di γ e` vera rispetto ad $I$ oppure il corpo di $\gamma$ non e` vero rispetto ad I.

Sia $ExP = <P, FS>$ un p.l.e. ed I un'interpretazione per ExP. I e` un *e–modello* per ExP se valgono le seguenti condizioni:

1. per ogni coppia $<p, [<A_{p,1}, D_{p,1}>,...,<A_{p,maxl(p)}, D_{p,maxl(p)}>]> \in FS$ abbiamo che:
   1.1 per ogni lev, $1 \le lev \le maxl(p)$, per ogni sostituzione θ, per ogni $\hat{t} \in A_{p,lev}$, tali che non esistono lev', $lev < lev' \le maxl(p)$, una sostituzione θ' ed una ennupla $t' \in D_{p,lev'}$, tali che $\hat{t}\theta = t'\theta'$, allora $p(\hat{t}\theta)$ e` vero rispetto ad I,
   1.2 per ogni lev, $1 \le lev \le maxl(p)$, per ogni sostituzione θ, per ogni $\hat{t} \in D_{p,lev}$, tali che non esistono lev', $lev < lev' \le maxl(p)$, una sostituzione θ' ed una ennupla $t' \in A_{p,lev'}$, tali che $\hat{t}\theta = t'\theta'$, allora $p(\hat{t}\theta)$ non e` vero rispetto ad I;
2. ogni istanza ground di una regola di P e` o vera rispetto ad $I$, oppure la sua testa e`, supponiamo, $p(\hat{a})$, ed esistono lev, $1 \le lev \le maxl(p)$ ed una sostituzione θ tali che per qualche ennupla $t \in D_{p,lev}$, si ha $t\theta = \hat{a}$.

Si noti che le condizioni 1.1. e 1.2. precedenti risultano essere sempre consistenti, per definizione di p.l.e.

Un p.l.e. $ExP = <P, FS>$ si dice p.l.e. di Horn se P e` un programma logico di Horn. Le proprieta` P1. e P2. dei programmi logici di Horn sono valide per i p.l.e. di Horn.

Proposizione 1. *Sia ExP un p.l.e. di Horn. Siano M_1 ed M_2 due e-modelli di ExP. Allora $M_1 \cap M_2$ e` un e-modello di ExP.* □

Corollario 1. *Sia ExP un p.l.e. di Horn. Sia Γ_{ExP} la classe di tutti gli e-modelli di ExP. Allora $LM_{ExP} = \bigcap_{M \in \Gamma_{ExP}} M$ e` il minimo e-modello di ExP.* □

Si consideri ora un programma logico P. Sia $R(P)$ l'insieme delle regole di P. Sia $F(p, P)$ l'insieme di tutte le ennuple di termini (non necessariamente ground) $(t_1,...,t_{\alpha(p)})$ tali che

$p(t_1,...,t_{\alpha(p)})$ e` un fatto di P. Si definisce il p.l.e. associato a P, $E(P)$, come segue. $E(P) = <R(P), FS>$, dove per ogni simbolo di predicato $p \in S_P$, la coppia $<p, [<F(p, P), \emptyset>]>$ appartiene a FS.

Si noti che se P e` un programma logico di Horn, allora $E(P)$ e` un p.l.e. di Horn. Nel seguito del lavoro faremo perciò riferimento solo a p.l.e. di Horn.

Dato un programma logico P, si puo` dimostrare che P ed $E(P)$ hanno lo stesso "significato".

Proposizione 2. *Sia P un programma logico. Allora si ha:*

1. $U_P = U_{E(P)}$ *e* $B_P = B_{E(P)}$;
2. $I \subseteq B_P$ *e` un modello di P se e solo se I e` un e-modello per* $E(P)$;
3. $LM_{E(P)} = LM_P$. □

E' facile dedurre dalla definizione di p.l.e. che solo quei p.l.e. in cui per ogni p $maxl(p) = 1$ e $D_{p,1} = \emptyset$, corrispondono a qualche programma logico. Vedremo nel seguito che, in generale, un p.l.e. corrisponde ad un programma logico su cui siano stati eseguiti degli aggiornamenti.

Dato un p.l.e. ExP, consideriamo un aggiornamento U in cui tutte le pre(post)-condizioni siano vuote, ad esempio $U = u_1,...,u_n$. Verra` definita di seguito una sequenza di p.l.e., che corrisponde alla sequenza delle applicazioni delle singole operazioni di aggiornamento $u_i$, $1 \le i \le n$, che appartengono ad $U$. L'ultimo elemento della sequenza di p.l.e. corrispondera`, dunque, alla esecuzione di U su ExP. Una volta definita tale sequenza, definiremo l'interpretazione $I_{U',P'}$ che caratterizza la semantica di un qualunque aggiornamento U' su un programma logico P'.

Si consideri, dunque, un p.l.e. $ExP = <P, FS>$ ed un aggiornamento U della forma $U = u_1,...,u_n$. Sia U_i il sotto-aggiornamento definito da $U_i = [u_1,...,u_i]$, $0 \le i \le n$ - U_0 denota l'aggiornamento nullo, [], ed $U_n = U$ -. Definiamo induttivamente la sequenza

$$\Sigma(U_0, ExP), \Sigma(U_1, ExP), \Sigma(U_2, ExP),...,\Sigma(U, ExP)$$

come segue. Sia $\Sigma(U_i, ExP) = <P, FS_i>$, per ogni i, $0 \le i \le n$.

1. $\Sigma(U_0, ExP) = ExP = <P, FS> = <P, FS_0>$.
2. $\Sigma(U_{i+1}, ExP) = <P, \Phi(u_{i+1}, FS_i)>$

dove la trasformazione Φ e` definita, per casi, come segue:

(Caso 1., $u_{i+1} = mta(p(t))$) Distinguiamo due sottocasi:

(Caso 1.1.) Non esiste una coppia del tipo $<p, L_p>$ in FS_i. Allora:

$$\Phi(u_{i+1},FS_i) = FS_i \cup \{<p, [<\{t\}, \emptyset>]>\}$$

(Caso 1.2.) Esiste una coppia del tipo $<p, L_p>$ in FS_i. Distinguiamo ulteriormente quattro sottocasi.

(Caso 1.2.1.) La ennupla t e` ground ed $A_{maxl(p)}$ e $D_{maxl(p)}$ contengono solo ennuple ground. Allora:

$$\Phi(u_{i+1},FS_i) = FS_i - \{<p, [...<A_{maxl(p)}, D_{maxl(p)}>]>\} \cup \{<p, [...<A_{maxl(p)} \cup \{t\}, D_{maxl(p)} - \{t\}>]>\}$$

(Caso 1.2.2.) L'insieme $A_{maxl(p)}$ contiene almeno una ennupla non ground. Allora:

$$\Phi(u_{i+1},FS_i) = FS_i$$

se esiste almeno una ennupla $t' \in A_{maxl(p)}$ che sussume t, altrimenti:

$$\Phi(u_{i+1},FS_i) = FS_i - \{<p, [...<A_{maxl(p)}, D_{maxl(p)}>]>\} \cup \{<p, [...<A_{maxl(p)} \cup \{t\}, D_{maxl(p)}>]>\}$$

(Caso 1.2.3.) L'insieme $D_{maxl(p)}$ contiene almeno una ennupla non ground. Allora:

$$\Phi(u_{i+1},FS_i) = FS_i - \{<p, [...<A_{maxl(p)}, D_{maxl(p)}>]>\} \cup \{<p, [...<A_{maxl(p)}, D_{maxl(p)}>, <\{t\}_{maxl(p)+1}, \emptyset_{maxl(p)+1}>]>\}$$

(Caso 1.2.4.) La ennupla t non e` ground e $A_{maxl(p)}$ e $D_{maxl(p)}$ contengono solo ennuple ground. Allora:

$$\Phi(u_{i+1},FS_i) = FS_i - \{<p, [...<A_{maxl(p)}, D_{maxl(p)}>]>\} \cup \{<p, [...<A_{maxl(p)}, D_{maxl(p)}>, <\{t\}_{maxl(p)+1}, \emptyset_{maxl(p)+1}>]>\}$$

(Caso 2. $u_{i+1} = mtr(p(t))$) Distinguiamo due sottocasi:

(Caso 2.1.) Non esiste in FS_i una coppia del tipo $<p, L_p>$. Allora:

$$\Phi(u_{i+1},FS_i) = FS_i \cup \{<p, [<\emptyset, \{t\}>]>\}$$

(Caso 2.2.) Esiste in FS_i una coppia del tipo $<p, L_p>$. Distinguiamo ancora una volta quattro sottocasi:

(Caso 2.2.1) La ennupla t e` ground ed $A_{maxl(p)}$ e $D_{maxl(p)}$ contengono solo ennuple ground. Allora:

$$\Phi(u_{i+1},FS_i) = FS_i - \{<p, [...<A_{maxl(p)}, D_{maxl(p)}>]>\} \cup \{<p, [...<A_{maxl(p)} - \{t\}, D_{maxl(p)} \cup \{t\}>]>\}$$

(Caso 2.2.2.) L'insieme $D_{maxl(p)}$ contiene almeno una ennupla non ground. Allora:

$$\Phi(u_{i+1},FS_i) = FS_i$$

se esiste almeno una ennupla $t' \in D_{maxl(p)}$ che sussume t, altrimenti:

$$\Phi(u_{i+1},FS_i) = FS_i - \{<p, [...<A_{maxl(p)}, D_{maxl(p)}>]>\} \cup \{<p, [...<A_{maxl(p)}, D_{maxl(p)} \cup \{t\}>]>\}$$

(Caso 2.2.3.) L'insieme $A_{maxl(p)}$ contiene almeno una ennupla non ground. Allora:

$$\Phi(u_{i+1},FS_i) = FS_i - \{<p, [...<A_{maxl(p)}, D_{maxl(p)}>]>\} \cup \{<p, [...<A_{maxl(p)}, D_{maxl(p)}>, <\emptyset_{maxl(p)+1}, \{t\}_{maxl(p)+1}>]>\}$$

(Caso 2.2.4.) La ennupla t non e` ground e $A_{maxl(p)}$ e $D_{maxl(p)}$ contengono solo ennuple ground. Allora:

$$\Phi(u_{i+1},FS_i) = FS_i - \{<p, [...<A_{maxl(p)}, D_{maxl(p)}>]>\} \cup \{<p, [...<A_{maxl(p)}, D_{maxl(p)}>, <\emptyset_{maxl(p)+1}, \{t\}_{maxl(p)+1}>]>\}$$

Le trasfomazione descritta in precedenza, trasforma p.l.e. in p.l.e., come dimostra la seguente proposizione.

Proposizione 3. *Sia ExP un p.l.e. ed* $U = u_1,...,u_n$ *un aggiornamento. Allora per ogni indice i,* $0 \le i \le n$, $\Sigma(U_i, ExP)$ *e` un p.l.e.* □

E' utile notare che la definizione data in precedenza, garantisce anche la *composizionalita`* della trasformazione associata ad un aggiornamento senza pre(post)-condizioni.

Proposizione 4. *Sia ExP un p.l.e. ed* $U = u_1,...,u_n$ *un aggiornamento. Allora per ogni indice j,* $0 \le j \le n$, *si ha:*

$$\Sigma(U, ExP) = \Sigma([u_{j+1},...,u_n], \Sigma(U_j, ExP)) \quad \square$$

Passiamo ora a considerare aggiornamenti in cui appaiano pre(post)-condizioni. Sia, dunque, $U = L_1, u_1, L_2,...,L_{n-1}, u_{n-1}, L_n$ un aggiornamento, e sia *ExP* un p.l.e. Definiamo la struttura $\Sigma'(U, ExP)$ - che, intuitivamente, corrispondera` al risultato dell'esecuzione di $U'$ su *ExP* -, come segue. Se per ogni indice $j$, $1 \le j \le n$, $L_j$ e` vero rispetto a $LM_{\Sigma([u_1,...,u_{j-1}], ExP)}$, allora:

$$\Sigma'(U, ExP) = \Sigma([u_1,...,u_{n-1}], ExP)$$

altrimenti:

$$\Sigma'(U, ExP) = ExP$$

Si noti che la definizione precedente e` ben posta, dal momento che, per la Proposizione 3., per ogni indice $j$, $1 \le j \le n$, $\Sigma([u_1,...,u_{j-1}], ExP)$ e` un p.l.e. e, conseguentemente, per il Corollario 1., esso ammette modello minimo. Si noti, inoltre, che per definizione e per la Proposizione 3., per ogni p.l.e. *ExP* e per ogni aggiornamento U, $\Sigma'(U, ExP)$ e` un p.l.e. Di conseguenza, ancora una volta per il Corollario 1., anche $\Sigma'(U, ExP)$ ammette modello minimo.

Sia P un programma logico ed U un aggiornamento che deve essere eseguito su P. Definiamo l'interpretazione $I_{U,P}$ che definisce la semantica dell'esecuzione di U su P come segue:

$$I_{U,P} = LM_{\Sigma'(U, E(P))}$$

Dalle considerazioni precedenti segue che anche quest'ultima definizione e` ben posta. I risultati presentati di seguito provano che la definizione appena data di $I_{U,P}$ e` corretta.

Proposizione 5. (Correttezza) *Sia P un programma logico ed* $U = L_1,u_1,...,u_{n-1},L_n$ *un aggiornamento. Se per ogni indice i,* $1 \le i \le n$, L_i *e` vero rispetto a* $I_{[u_1,...,u_{i-1}],P}$, *allora per ogni indice j,* $1 \le j \le n-1$, $u_j$ *e` realizzato in* $I_{[L_1,u_1,...,u_j,L_{j+1}],P}$. □

Proposizione 6. (Non sovrapposizione) *Sia P un programma logico e* $U = L_1,u_1,...,u_{n-1},L_n$ *un aggiornamento e si assuma che per ogni indice i,* $1 \le i \le n$, L_i *e` vero rispetto a* $I_{[u_1,...,u_{i-1}],P}$. *Allora per ogni indice j,* $1 \le j \le n-1$, $u_j$ *e` realizzato da* $I_{U,P}$ *se l'aggiornamento* $[u_{j+1},...,u_n]$ *non contiene occorrenze di* $\ddot{u}_j$, *dove* $\ddot{u}_j$ *e` l'aggiornamento singolo ottenuto come opposto di* $u_j$ *(cioe`,* $\ddot{u}_j = mtr(p(\hat{a}))$ *se* $u_j = mta(p(\hat{a}))$, *e viceversa).* □

Un ultima considerazione riguarda il trattamento di sequenze di aggiornamenti, ed il problema della definizione del concetto di risposta corretta in quest'ambito.

La semantica associata ad una sequenza di aggiornamenti del tipo $U_1,U_2,...,U_m$ eseguiti su un programma logico P puo` essere definita come il minimo modello di $\Sigma'(U_m,\Sigma'(U_{m-1},...\Sigma'(U_1, E(P))...)$. Ora, dato un goal $\leftarrow G$ da eseguire su un programma logico *UP* su cui sia stata eseguita una sequenza $H = U_1,U_2,...,U_m$ di aggiornamenti - H rappresenta la "storia" del programma -, si puo` definire risposta corretta una qualunque sostituzione $\theta$ tale che $G\theta$ e` vero rispetto ad $LM_{\Sigma'(U_m,\Sigma'(U_{m-1},...\Sigma'(U_1, E(UP))...)}$.

Con i risultati precedenti si chiude questo paragrafo. In appendice e` riportato un esempio "completo" del calcolo dell'interpretazione risultato di un aggiornamento su un programma logico. Nel prossimo paragrafo verranno presentate alcune conclusioni sull'argomento trattato in questo lavoro, e discusse alcuni possibili sviluppi futuri di questa ricerca.

5. Conclusioni.

In questo lavoro e` stata definita una semantica dichiarativa formale per alcune operazioni di aggiornamento sui programmi logici di Horn. Tali operazioni corrispondono ad aggiornamenti specificati come goal posti ad un programma logico, che richiedano inserzioni o cancellazioni di atomi. E' stato posto inoltre un vincolo sulla sintassi delle operazioni di aggiornamento che garantisce la deterministicita` della semantica associata alla loro esecuzione. La semantica proposta corrisponde ad un concetto - che noi crediamo essere abbastanza intuitivo - di realizzazione minima di un aggiornamento, e che fa riferimento all'insieme delle conseguenze logiche "correnti" del programma in oggetto, e non, come nel caso delle operazioni di aggiornamento del Prolog, al programma vero e proprio. Questo consente di evitare alcuni risultati controintuitivi che gli interpreti Prolog a volte producono come effetti di operazioni di aggiornamento. Al fine di evidenziare similitudini e differenze tra l'approccio qui proposto ed il Prolog, sono stati usati un certo numero di esempi comparativi.

Alcuni problemi rimangono aperti. Tra questi vogliamo ricordare: (1) semantica dichiarativa degli aggiornamenti non-deterministici, (2) semantica dichiarativa di inserzioni o cancellazioni di regole, (3) semantica dichiarativa di programmi logici che contengano regole nei cui corpi siano specificate operazioni di aggiornamento, (4) semantica operazionale dei programmi logici su cui siano stati eseguiti aggiornamenti e/o che contengano regole in cui compaiano aggiornamenti, (5) tecniche di ottimizzazione.

Ringraziamenti. L'autore ringrazia Nicola Leone e Geraldine Scalzo per gli utili commenti sulle versioni preliminari di questo lavoro.

6. Riferimenti.

[1] W.F. Clocksin, and C.S. Mellish, "Programming in Prolog", Springer-Verlag, Berlin, 1984.

[2] J.W. Lloyd, "Foundation of Logic Programming", Springer-Verlag, New York, NY, 1987.

[3] R. Fagin, G. Kuper, J.D. Ullman and M.Y. Vardi, "Updating logical databases", Advances in Comp.Res., vol.3, 1-18, JAI Press Inc., 1986.

[4] M. Winslett, "A model-based approach to updating databases with incomplete information", ACM TODS, vol.13, No.2, 167-196, 1988.

[5] S. Manchanda, and D.S. Warren, "Towards a logical theory of database view updates", Int.Worksh. on Foundations of Deductive databases and Logic Programming, J.Minker ed.,

Aug. 1988.

[6] S. Manchanda, "Declarative expression of deductive database updates", *Proc. ACM PODS*, 1989.

[7] Rossi,F., and S.Naqvi, "Contribution to the view update problem" *Proc. Int. Conf. Logic Programming* 1989, 388-415.

[8] Warren, D.S., "Database updates in pure Prolog", *Proc. Int. Conf. on 5th Generation Comp. Syst.*, ICOT, 1984.

Appendice.

Di seguito e` riportato il calcolo dell'interpretazione che caratterizza la semantica dichiarativa di un aggiornamento significativamente complesso su un programma logico di Horn.

Esempio 7. Si consideri il seguente programma logico P.

$s(X) \leftarrow q(X), p(X)$

$p(X) \leftarrow r(X), q(X)$

$p(a).$

$r(b).$

$r(c).$

$q(b).$

$q(a).$

Si supponga di volere eseguire su P il seguente aggiornamento:

$U = s(X), mtr(p(Y)), \neg s(b), mta(p(d)), mta(s(d))$

Calcoliamo, innanzitutto, le sequenza di p.l.e. associata all'aggiornamento incondizionato $mtr(p(Y)), mta(p(d)), mta(s(d))$.

1. $\Sigma([], E(P)) = E(P) = \langle P', FS_0 \rangle = E_0$

dove:

$P' = \{[s(X) \leftarrow q(X), p(X)], [p(X) \leftarrow r(X), q(X)]\}$

$FS_0 = \{\langle q, [\langle \{a, b\}, \emptyset \rangle] \rangle, \langle r, [\langle \{b, c\}, \emptyset \rangle] \rangle$
$\langle p, [\langle \{a\}, \emptyset \rangle] \rangle, \langle s, [\langle \emptyset, \emptyset \rangle] \rangle\}$

2. $\Sigma([mtr(p(Y))], E(P)) = \langle P', FS_1 \rangle = E_1$

dove:

$FS_1 = \{\langle q, [\langle \{a, b\}, \emptyset \rangle] \rangle, \langle r, [\langle \{b, c\}, \emptyset \rangle] \rangle$
$\langle p, [\langle \{a\}, \emptyset \rangle, \langle \emptyset, \{Y\} \rangle] \rangle, \langle s, [\langle \emptyset, \emptyset \rangle] \rangle\}$

3. $\Sigma([mtr(p(Y)), mta(p(d))], E(P)) = \langle P', FS_2 \rangle = E_2$

dove:

$FS_2 = \{\langle q, [\langle \{a, b\}, \emptyset \rangle] \rangle, \langle r, [\langle \{b, c\}, \emptyset \rangle] \rangle$
$\langle p, [\langle \{a\}, \emptyset \rangle, \langle \emptyset, \{Y\} \rangle, \langle \{d\}, \emptyset \rangle] \rangle, \langle s, [\langle \emptyset, \emptyset \rangle] \rangle\}$

4. $\Sigma([mtr(p(Y)), mta(p(d)), mta(s(d))], E(P)) = \langle P', FS_3 \rangle = E_3$ dove:

$FS_2 = \{\langle q, [\langle \{a, b\}, \emptyset \rangle] \rangle, \langle r, [\langle \{b, c\}, \emptyset \rangle] \rangle$
$\langle p, [\langle \{a\}, \emptyset \rangle, \langle \emptyset, \{Y\} \rangle, \langle \{d\}, \emptyset \rangle] \rangle, \langle s, [\langle \{d\}, \emptyset \rangle] \rangle\}$

Consideriamo ora la soddisfacibilita` delle condizioni che appaiono in U. A tal fine calcoliamo i minimi modelli di Herbrand associati a E_0, E_1.

$LM_{E_0} = LM_P = \{q(a), q(b), r(b), r(c), p(a), p(b), s(b)\}$

$LM_{E_1} = \{q(a), q(b), r(b), r(c)\}$

Ora, U produce effetto non nullo su P solo se esistono due sostituzioni θ_1 ed θ_2 tali che $s(X)\theta_1$ e` vero rispetto a $LM_{E_0}$ e $\neg s(b)\theta_2$ e` vero rispetto a LM_{E_1}. La condizione e` verificata per:

$\theta_1 = \{X/b\}; \qquad \theta_2 = \emptyset;$

Di conseguenza otteniamo:

$I_{U,P} = LM_{E_3} = \{q(a), q(b), r(b), r(c), p(d), s(d)\}$

MONOTONICITY IN AE THEORIES: PRELIMINARY REPORT

Piero A. Bonatti

Abstract: In this paper, the basic definitions for auto-epistemic logics are rewritten by exploiting transformations on sets of statements - called *A-transforms* - that are implicit in the fix-point definition of stable expansions. There are at least two advantages in making such transformations explicit: first, A-transforms allow more symmetrical definitions; secondly, by posing monotonicity constraints on A-transforms, certain classes of auto-epistemic theories, whose stable expansions enjoy interesting properties, can be naturally characterised. These results are then applied to the semantics of negation in Logic Programming and correspondence between converging A-transforms and a generalised form of stratified programs is shown.

1. Introduction

Many of the technical problems that have to be faced when dealing with auto-epistemic theories are rooted in the non-constructive character of their stable expansions. There seems to be nothing like a monotonic operator that will produce the desired expansion when iteratively composed with itself. This work is a first attempt at overcoming this kind of problem, based on a notion of *weak monotonicity* for an operator that is implicit in the fix-point equation characterising stable expansions. Analogous (classically) monotonic operators already appeared in [Marek et al. 89], but they generate expansions, rather than *stable* expansions. With our approach, instead - based upon a pre-ordering on sets of formulas - stable expansions can be related with the operator's iterative application, providing a more "operational" characterisation of an interesting class of autoepistemic theories. In the next section a brief summary of auto-epistemic logics will be given. Section three contains the main results, while section four shows one of their possible applications in the field of logic programming.

2. Auto-epistemic logics revisited

Let us recall the usual definitions for auto-epistemic logics, that can be found - for example - in [Moore 85]. The language is a propositional modal language $\mathcal{L}$,