ELSEVIER
Theoretical Computer Science 149 (1995) 201–229
Theoretical Computer Science

Fully abstract compositional semantics for an algebra of logic programs

Antonio Brogi, Franco Turini*
Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy

Received September 1993; revised November 1994
Communicated by G. Levi

Abstract

A simple extension of logic programming consists of introducing a set of basic program composition operations, which form an algebra of logic programs with interesting properties for reasoning about programs and program compositions. From a programming perspective, the operations enhance the expressive power of the logic programing paradigm by supporting a wealth of programming techniques, ranging from software engineering to artificial intelligence applications.

This paper focuses on the semantics of program composition operations. It is shown that the immediate consequence operator $T(P)$ properly characterises the intended meaning of a program P when considering compositions of programs. More precisely, it is shown that the $T(P)$ semantics is both compositional and fully abstract w.r.t. the set of composition operations of the algebra. This implies that the $T(P)$ semantics induces the coarsest equivalence relation on programs (subsumption-equivalence) and that any other semantics of programs must induce the same equivalence relation to be both compositional and fully abstract w.r.t. the whole set of operations of the algebra. The $T(P)$ semantics is then related to other well known semantics for logic programs which induce coarser equivalence relations. In particular, three equivalence relations, originally studied by Maher (1988), are considered: Weak subsumption-equivalence, logical equivalence and least Herbrand model equivalence. It is shown that the chain of equivalence relations composed by weak subsumption-equivalence, logical equivalence and least Herbrand model equivalence coincides with the chain of fully abstract compositional equivalence relations for proper subsets of the operations of the algebra, obtained by dropping one operation at a time from the set of compositions.

1. Introduction

Logic programming is considered a promising candidate for the definition of a computational logic capable of dealing with different aspects of computing. To this end, several extensions of the basic formalism of Horn clauses have been proposed to

*Corresponding author. E-mail: {brogi, turini}@di.unipi.it.

SSDI 0304-3975(95)00012-7

improve its knowledge representation and problem solving capabilities. Some of these extensions are aimed at supporting enhanced kowledge representation and reasoning capabilities, which are essential for typical artifical intelligence applications such as nonmonotonic reasoning. Other extensions are instead aimed at providing enhanced capabilities for structuring and modularising programs, which are essential in a software engineering perspective where the issues of incremental program development, modularity and object-orientation are of primary concern.

In previous work [3, 5], we have studied a simple extension of logic programming which consists of introducing a set of basic program composition operations. The operations are defined in a semantics-driven style, following the observation that if the meaning of a program P is denoted by the corresponding *immediate consequence operator* $T(P)$ then such a meaning is a homomorphism for several interesting operations on programs. From a programming perspective, these operations enhance the expressive power of the logic programming paradigm by supporting a wealth of programming applications, ranging from software engineering techniques for structuring and modularising programs [6, 8, 9] to artificial intelligence techniques for hypothetical and default reasoning [5, 7]. A thorogh discussion of the applications of program composition operations can be found in [3].

The choice of denoting the meaning of a program P by the corresponding immediate consequence operator $T(P)$ therefore permits to define the semantics of program compositions in a compositional (viz. homomorphic) way, that is to define the semantics of a composition of programs in terms of the semantics of the separate programs. The adoption of the $T(P)$ semantics, however, induces the following equivalence relation on logic programs: Two programs are equivalent if and only if they have the same function $T(P)$, that is, their immediate consequence operators coincide on every Herbrand interpretation. Maher [17] gave an alternative formulation of this equivalence relation, and showed that the syntactic notion of *subsumption-equivalence* coincides with the equality of functions $T(P)$ on programs.

The equivalence relation induced by the $T(P)$ semantics is however quite fine. For instance, consider the program representing the natural numbers

$$nat(zero) \leftarrow$$

$$nat(s(x)) \leftarrow nat(x)$$

and extend it with an axiom explicitly stating that the successor of zero is a natural number, that is,

$$nat(s(zero)) \leftarrow$$

$$nat(zero) \leftarrow$$

$$nat(s(x)) \leftarrow nat(x)$$

It is easy to observe that the two programs do not have the same immediate consequence operator (consider the empty interpretation). Therefore, in this case, the

$T(P)$ semantics does not consider equivalent a program and its extension with one of its logical consequences.

A natural and intriguing question is whether or not it is possible to choose a different denotation of programs, which retains the compositionality property of the $T(P)$ semantics while inducing a coarser equivalence relation on programs. This problem can be formally addressed by studying the property of full abstraction in the semantics of programming languages. Informally speaking, the property of compositionality states that equivalent programs are indistinguishable, that is, they exhibit the same observable operational behaviour in any possible context. The property of full abstraction, on the other hand, states that indistinguishable programs are equivalent.

In this paper, we show that the $T(P)$ semantics is fully abstract with respect to the observable behaviours of programs for the set of operations of the algebra. More precisely, if two programs exhibit equal behaviours in any context of program compositions then the two programs have the same immediate consequence operator $T(P)$.

The former result provides the choice of denoting a program P with the corresponding immediate consequence operator $T(P)$ with a firm mathematical justification. Indeed the property of full abstraction states that, for the set of compositions considered, the $T(P)$ semantics induces the coarsest equivalence relation on programs, and that therefore any other possible denotation of programs must induce the same equivalence relation to be both compositional and fully abstract.

The $T(P)$ semantics is then related to other semantics for logic programs which induce coarser equivalence relations. Three equivalence relations, originally studied by Maher [17], are considered: Weak subsumption-equivalence, logical equivalence and least Herbrand model equivalence. It is shown that each of these equivalence relations is both compositional and fully abstract for proper subsets of the set of operations of the algebra, obtained by dropping one operation at a time from the set of compositions.

The paper is organized as follows. The algebra of logic programs is introduced in Section 2, where a motivating example of use of the operations is also illustrated. The properties of compositionality and full abstraction are formally defined in Section 3. The full abstraction of the equivalence relation induced by the $T(P)$ semantics (subsumption-equivalence) is proved in Section 4. Section 5 is devoted to investigate the relations with other semantics. Finally, Section 6 contains a summary of the results, the study of other sets of compositions and some concluding remarks.

2. An algebra of logic programs

This section introduces the set of composition operations forming the algebra of logic programs, originally defined in [3, 18]. An example of use of the operations for programming is also illustrated.

2.1. Program composition operations

Four basic operations for composing logic programs are introduced, following [3, 18]: Encapsulation (denoted by $*$), union ($\cup$), intersection ($\cap$), and import ($\lhd$). The operations are defined in a semantics-driven style, following the intuition that if the meaning of a proram P is denoted by the corresponding *immediate consequence operator* $T(P)$, then such a meaning is a homomorphism for several interesting operations on programs. Notice that the standard least Herbrand model semantics of logic programming is not appropriate to model compositions of programs in that it does not enjoy the compositionality requirement. Actually the least Herbrand model of a program cannot be obtained, in general, from the least Herbrand models of its clauses. The idea of employing the immediate consequence operator to denote the meaning of logic programs was originally proposed by Mancarella and Pedreschi [18] building on previous work by Lassez and Maher [15], O'Keefe [20] and Fitting [12]. We adhere to this approach and denote each program P with the corresponding $T(P)$.

Recall that, for a logic program P, the immediate consequence operator $T(P)$ is a continuous mapping over Herbrand interpretations defined as follows [22]. For any Herbrand interpretation I:

$$A \in T(P)(I) \Leftrightarrow (\exists \bar{B}: A \leftarrow \bar{B} \in ground(P) \wedge \bar{B} \subseteq I),$$

where $\bar{B}$ is a (possibly empty) conjunction of atoms. The powers of $T(P)$ are defined as usual [1]:

$$T^0(P)(I) = I,$$

$$T^{n+1}(P)(I) = T(P)(T^n(P)(I)),$$

$$T^{\omega}(P)(I) = \bigcup_{n<\omega} T^n(P)(I).$$

and $T^{\alpha}(P)(\emptyset)$ is abbreviated to $T^{\alpha}(P)$.

The semantics of program compositions is given in a compositional way by extending the definition of T with respect to the first argument. We assume that the language in which programs are written is fixed. Namely, the Herbrand base we refer to is determined by a set of function and predicate symbols that include all function and predicate symbols used in the programs being considered.

For any Herbrand interpretation I:

$$T(P^*)(I) = T^{\omega}(P),$$

$$T(P \cup Q)(I) = T(P)(I) \cup T(Q)(I),$$

$$T(P \cap Q)(I) = T(P)(I) \cap T(Q)(I),$$

$$T(P \lhd Q)(I) = T(P)(I \cup T^{\omega}(Q)).$$

The above definition generalises the notion of immediate consequence operator from programs to compositions of programs. In particular, for any interpretation I,

the formulae which may be derived in the encapsulated program P^* are all the formulae which may be derived from P in an arbitrary (finite) number of steps. The operations of union and intersection of programs directly relate to their set-theoretic equivalent. The set of immediate consequences of the union (resp. intersection) of two programs is the set-theoretic union (resp. intersection) of the sets of immediate consequences of the separate programs. Finally, for any interpretation I, the set of immediate consequences of the import of two programs $P \triangleleft Q$ is the set of formulae which may be derived in the importing program P in a single deduction step from I and from the set of formulae which may be derived in the imported program Q in an arbitrary (finite) number of steps.

The operations $*$, $\cup$, $\cap$ and $\triangleleft$ satisfy a number of algebraic properties such as associativity, commutativity and distributivity. The resulting algebra [3] extends the algebra presented in [18] and offers a formal basis for proving properties of program compositions. For instance, syntactically different program compositions may be compared and simplified by means of the properties of program composition operations.

An alternative definition of program composition operations can be given by characterising the operational behaviour of program compositions. The operational behaviour of logic programs is usually given by means of the success set $SS(P)$ of a program P, that is the set of ground atoms A such that $P \cup \{\leftarrow A\}$ has a SLD refutation [1, 16, 22]. A simple way of characterising the operational behaviour of compositions of programs is to directly extend the definition of SLD refutation to deal with program compositions. The standard SLD derivation relation may be defined by means of inference rules of the form

$$\frac{\textit{Premise}}{\textit{Conclusion}}$$

asserting that *Conclusion* holds whenever *Premise* holds. We write $P \vdash G$ if there exists a SLD refutation for a goal G in a program P.

$$\frac{}{P \vdash \textit{empty}} \tag{1}$$

$$\frac{P \vdash G_1 \ \wedge \ P \vdash G_2}{P \vdash (G_1, G_2)} \tag{2}$$

$$\frac{P \vdash (A \leftarrow G) \ \wedge \ P \vdash G}{P \vdash A} \tag{3}$$

Rule (1) states that the empty goal, denoted by *empty*, is solved in any program P. Rule (2) deals with conjunctive goals. It states that a conjunction (G_1, G_2) is solved in a program P if G_1 is solved in P and G_2 is solved in P. Finally, rule (3) deals with atomic goal reduction. To solve an atomic goal A, choose a clause from program P and recursively solve the body of the clause in P. Notice that, for the sake of

simplicity, substitutions are omitted in that we are interested in characterising only the (ground) success set of a program.

Program clauses are represented by means of the following rule:

$$\frac{P \text{ is a plain program} \ \wedge \ A \leftarrow G \in ground(P)}{P \vdash (A \leftarrow G)} \tag{4}$$

It is easy to show that for any program P and for any atomic formula A:

$$A \in SS(P) \ \Leftrightarrow \ P \vdash A.$$

The derivation relation $\vdash$ can be generalized to the case of program compositions in a simple way. Namely, each composition operation is modelled by adding new inference rules to rules (1)–(4).

$$\frac{P \vdash (A \leftarrow G)}{P \cup Q \vdash (A \leftarrow G)} \tag{5}$$

$$\frac{Q \vdash (A \leftarrow G)}{P \cup Q \vdash (A \leftarrow G)} \tag{6}$$

$$\frac{P \vdash (A \leftarrow G_1) \ \wedge \ Q \vdash (A \leftarrow G_2)}{P \cap Q \vdash (A \leftarrow G_1, G_2)} \tag{7}$$

$$\frac{P \vdash A}{P^* \vdash (A \leftarrow empty)} \tag{8}$$

$$\frac{P \vdash (A \leftarrow G_1, G_2) \ \wedge \ Q \vdash G_2}{P \lhd Q \vdash (A \leftarrow G_1)} \tag{9}$$

Rules (5) and (6) state that a clause $A \leftarrow G$ belongs to the program expressions $P \cup Q$ if it belongs either to P or to Q. Rule (7) states that a clause $A \leftarrow G$ belongs to $P \cap Q$ if there is a clause $A \leftarrow G_1$ in P and there is a clause $A \leftarrow G_2$ in Q such that $G = (G_1, G_2)$. Rule (8) states that the program expression P^* contains a unit clause $A \leftarrow empty$ for each atom A which is provable in P. Finally, rule (9) deals with the import operation. It states that the clauses in $P \lhd Q$ are obtained from the clauses in P by dropping the calls to Q, provided that they are provable in Q.

The extended derivation relation $\vdash$ defined by rules (1)–(9) characterises the operational behaviour of arbitrary compositions of programs. The equivalence of the operational and of the denotational semantics of program compositions is stated by the following proposition.

Proposition 1 (Brogi [3]). *For any program expression P and any atomic formula A:*

$$P \vdash A \ \Leftrightarrow \ A \in T^{\omega}(P).$$

In [3], two other operational characterisations of program compositions are given and proved to be equivalent to the denotational semantics. A compilation-oriented

implementation is obtained by means of a program transformation ρ which maps any composition of programs P into an equivalent plain program $\rho(P)$ such that the success set of the transformed program $\rho(P)$ coincides with the least fixpoint of $T(P)$. An interpretation-oriented implementation is obtained by means of an extended "vanilla" metainterpreter which realises the various compositions through metalevel compositions of object level programs. For any program composition P, the set of sentences provable in the corresponding extended vanilla metainterpreter coincides with the least fixpoint of $T(P)$. The definition of the operational semantics of program expressions by means of inference rules, however, may be easier to read than the transformational definition, which involves a renaming mechanism, and more general than the metalogical definition, which relies on metaprogramming.

As pointed out in the introduction, this language of program expressions, albeit simple, finds natural applications in several domains, ranging over expert systems, hypothetical and hierarchical reasoning, knowledge assimilation and modularisation. An example of such applications is illustrated in the next section.

2.2. *A motivating example*

We show how the previously introduced operations can be used to address one of the hot issues in the logic programming field, that is how to provide logic programming with a useful and, at the same time, logic-based modular structure. A more thorough presentation of these ideas can be found in [3, 8].

Traditional modular programming requires that a module can import from another one only its functionality without caring of the implementation. For example, if a module needs a *sort* operation over lists, the modular style requires the importation of a *sort* relation from another module, which is free to implement it according to any of the sorting algorithms. This kind of behaviour can be esily realised in logic programming by exploiting the operations $\cup$ and $*$. If P is the "main" program and S is the module implementing the *sort* relation, the combined program is expected to behave as

$$P \cup S^*.$$

The usefulness of the operation $\cap$ has been shown elsewhere in the context of knowledge representation and reasoning [5, 9]. Here we show that, in the context of modularising logic programs, it can be used to restrict import/export operations to a set of pre-selected predicates. Given a set of predicates π, let π itself denote the program: $\{p(x) \leftarrow \mid p \in \pi\}$. Now, the importation of only the extensional definitions of predicates in π from a module Q to a module P is easily defined as

$$P \cup (\pi \cap Q^*).$$

However, the very nature of logic programming allows a more refined definition of module composition. Indeed, while in traditional languages the objects which can be imported/exported are either data or functions/procedures, in logic programming we can distinguish between either importing/exporting a whole predicate definition or

part of it, that is some of its clauses. Even more drastically, we can think that a predicate definition is in general spread over several different modules.

By exploiting this possibility, we obtain several forms of information hiding, based on the observation that the knowledge about a procedure is available at two different levels: the *intensional* and the *extensional* level. The former is (an abstraction of) the code of a predicate (i.e. the clauses defining it). The latter is the set of atomic formulae provable for that predicate. The exportation at the extensional level is the one which is supported by conventional modular languages, and the operations $\cup$, $\cap$ and $*$ provide suitable mechanisms for dealing with it.

The operation $\lhd$ allows us to exploit the potential flexibility of logic programming with respect to importing/exporting knowledge. In its basic usage, the $\lhd$ operation builds a module $P \lhd Q$ out of a pair of modules P and Q, which play the role of the *visible* and *hidden* part of the module, respectively. Indeed, several forms of information hiding/exporting can be accommodated by exploiting the operation $\lhd$.

- *Full hiding*: A predicate is defined only in the hidden part Q of a module $P \lhd Q$. In such a way, the predicate is exported neither at the extensional nor at the intensional level.
- *Implementation hiding*: A predicate is defined in the visible part P of a module $P \lhd Q$, but its actual implementation is entirely in the hidden part Q. In such a way the predicate is exported only at the extensional level.
- *Partial visibility*: A predicate is defined partly in the visible part P and partly in the hidden part Q of a module $P \lhd Q$. In such a way the predicate is partly exported at the intensional level.
- *Full visibility*: A predicate is defined in the visible part P of a module $P \lhd Q$. In such a way the predicate is exported both at the intensional and at the extensional level.

As an example of use of $\lhd$, consider the following simple situation. Suppose that a bank is supported by an expert system for granting credits to customers. One of the procedures of the expert system is about judging the reliability of customers, and this may involve some information about the customers which a bank manager would prefer to keep hidden. This can be achieved by implementing the expert system through a module expression in which the intension of the critical predicates is partly hidden. In what follows the programs *Rules*, *Public* and *Private* represent the basic expert system rules, the public domain database information and the critical expert system knowledge, respectively.

Rules

gives_credit$(x) \leftarrow$
  good_customer(x)
good_customer$(x) \leftarrow$
  customer(x)
  has_account(x, t)
  threshold(y)
  greater_than(t, y)

Public

customer(*smith*) $\leftarrow$
customer(*lewis*) $\leftarrow$
has_account(*lewis*, ...) $\leftarrow$

Private

good_customer$(x) \leftarrow$
  in_politics(x)
has_account(*smith*, ...) $\leftarrow$
in_politics(*brown*) $\leftarrow$
threshold(...) $\leftarrow$

The overall expert system can then be obtained by the module:

$$GrantCredits = (Rules \triangleleft Private) \cup Public$$

Notice that in this way neither the extension nor the intension of the *Private* component is visible to the external world.

3. Compositionality and full abstraction

A semantics for a programming language provides meanings for programs or, more generally, program parts. Moreover, each method of giving semantics to a programming language induces an equivalence relation on programs. Namely, two programs are equivalent if and only if they have the same meaning in the chosen semantics.

The properties of *compositionality* and *full abstraction* have been recognised as two fundamental concepts in the studies on the semantics of programming languages [19,21]. Roughly, a semantics is compositional if equivalent programs (or program parts) are indistinguishable, that is, if they "exhibit equal observable behaviours in all possible contexts". On the other hand, a semantics is fully abstract if indistinguishable programs (or program parts) are equivalent.

Both these properties refer to a notion of observable behaviours of programs and to a notion of possible compositions for programs. The former can be represented by a mapping $Ob(\cdot)$ which associates an object $Ob(P)$ denoting the observable behaviours with every program P. The latter can be represented by a set Com of (possibly partial) functions over programs. A semantics is compositional if the induced equivalence is a congruence for the pair (Ob, Com), that is, if it preserves the observables and is a congruence for the set of compositions. There is always a coarsest congruence for (Ob, Com), which is intuitively the "indistinguishability relation". A semantics is fully abstract if the induced equivalence includes this largest congruence. A semantics is both compositional and fully abstract if it coincides with it.

We now formally introduce these notions according to [13]. Let Pr be a class of programs, and let Ob be a mapping which associates with every program P an object $Ob(P)$ denoting the observable behaviours of P. Moreover, let Com be a class of (possibly partial) functions over Pr, called program compositions. For every n-ary $\mathscr{F} \in Com$ and every $P_1, \ldots, P_n \in Pr$, if $\mathscr{F}(P_1, \ldots, P_n)$ is defined then $\mathscr{F}(P_1, \ldots, P_n) \in Pr$.

An equivalence relation $\equiv$ over Pr *preserves* Ob iff $\forall P, Q \in Pr$:

$$P \equiv Q \Rightarrow Ob(P) = Ob(Q)$$

An equivalence relation $\equiv$ over Pr is *congruence* for Com iff $\forall P_1, \ldots, P_n$, $Q_1, \ldots, Q_n \in Pr$, $\forall \mathscr{F} \in Com$:

$$P_i \equiv Q_i \ (i = 1, \ldots, n) \Rightarrow \mathscr{F}(P_1, \ldots, P_n) \equiv \mathscr{F}(Q_1, \ldots, Q_n).$$

A *congruence* for (Ob, Com) is a congruence for Com which preserves Ob. Such a congruence is called *compositional*.

Two programs P and Q are *distinguishable* under (Ob, Com) if there exists a context (defined via Com) in which the substitution of P with Q changes the external behaviour (defined via Ob) of the context. Formally, let Com^c be the closure of Com under (partial functions) compositions, including the identity function. Then two programs P and Q are *distinguishable* if

$$\exists \mathscr{F} \in Com^c \ \exists P_1, \ldots, P_n \in Pr: \quad Ob(\mathscr{F}(P, P_1, \ldots, P_n)) \neq (Ob(\mathscr{F}(Q, P_1, \ldots, P_n)).$$

We put

$$P \cong Q \Leftrightarrow P \text{ and } Q \text{ are not distinguishable under } (Ob, Com).$$

Finally, an equivalence relation $\equiv$ is *fully abstract* if and only if $\forall P, Q \in Pr$:

$$P \cong Q \Rightarrow P \equiv Q.$$

In order to study the properties of compositionality and full abstraction for the algebra of logic programs, the set of programs Pr, the set of compositions Com and the set of observables Ob have to be fixed.

The set Com is the set of program composition operations that were introduced in Section 2, that is,

$$Com = \{*, \cup, \cap, \lhd\}.$$

The set Pr is the set of program expressions, that is the set of definite logic programs possibly composed via elements of Com. Notice that all the compositions are total functions over Pr.

The observable behaviours of a program may be defined in different ways, depending on which aspects of the computation one is interested in looking at. In the case of logic programs, a natural choice of the observables is the *success set* of a program P [1, 16, 22]. Therefore we put

$$Ob(P) = SS(P).$$

Notice that the definition of indistinguishability requires the notion of success set (viz. observables) to be extended from programs to arbitrary compositions of programs, as already discussed in Section 2. In the rest of the paper, the equivalence $SS(P) = T^{\omega}(P)$ (Proposition 1) will be exploited in the proofs. We will analyse fully abstract equivalence relations for different subsets of $Com = \{*, \cup, \cap, \lhd\}$ under the same set of observables Ob. To highlight the set of compositions considered, we will denote by $\cong_X$ the indistinguishability relation over (Ob, X) for $X \subseteq Com$.

4. Subsumption-equivalence

The algebra of logic programs defined in Section 2 relies on the choice of denoting the meaning of a program P by the corresponding immediate consequence operator $T(P)$. Such a denotation induces the following equivalence relation, denoted by $\equiv_T$,

on programs. Two programs are equivalent if and only if the corresponding immediate consequence oprators coincide on every Herbrand interpretation.

Definition 1. The equivalence relation $\equiv_T$ on Pr is defined as follows. Let $P, Q \in Pr$, then

$$P \equiv_T Q \Leftrightarrow T(P) = T(Q).$$

Maher [17] first studied the equivalence relation induced by the $T(P)$ semantics. He introduced a syntactic notion of equivalence, *subsumption-equivalence*, and showed that it corresponds to the equality of functions $T(P)$ on programs. Let C_1 and C_2 be the definite clauses $A \leftarrow \bar{B}$ and $D \leftarrow \bar{E}$, respectively. C_1 is subsumed by C_2 if there is a substitution ϑ such that $A = D\vartheta$ and $\bar{E}\vartheta \subseteq \bar{B}$. Two logic programs P and Q are subsumption equivalent if every clause of P is subsumed by some clause of Q and vice versa. Existing algorithms [14] can be therefore exploited to determine whether two programs are $T(P)$ equivalent.

It is easy to show that the equality induced by the $T(P)$ semantics is a congruence for $(Ob, \{*, \cup, \cap, \lhd\})$. Indeed the $T(P)$ semantics of the operations $*$, $\cup$, $\cap$ and $\lhd$ is defined in a homomorphic way, according to [3, 18]. Namely, the meaning of a composition of programs (e.g. $T(P \cup Q)$) is determined by the meaning of the separate programs ($T(P)$ and $T(Q)$). Furthermore, it is easy to observe that if two programs are $T(P)$ equivalent then they also have the same observable behaviour.

Proposition 2. *$\equiv_T$ is a congruence for $(Ob, \{*, \cup, \cap, \lhd\})$.*

Proof. (1) $\equiv_T$ preserves Ob: Obvious, since $\forall P, Q \in Pr$: $P \equiv_T Q \Leftrightarrow \forall I$: $(T(P)(I) = T(Q)(I))$. This implies, by definition of the powers of T, that $T^\omega(P) = T^\omega(Q)$ and hence, by definition of Ob, that $Ob(P) = Ob(Q)$.

(2) $\equiv_T$ is a congruence for $\{*, \cup, \cap, \lhd\}$: We have to prove that $\forall P_1, P_2, Q_1, Q_2 \in Pr$, if $P_1 \equiv_T Q_1$ and $P_2 \equiv_T Q_2$ then

(i) $P_1^* \equiv_T Q_1^*$,

(ii) $P_1 \cup P_2 \equiv_T Q_1 \cup Q_2$,

(iii) $P_1 \cap P_2 \equiv_T Q_1 \cap Q_2$,

(iv) $P_1 \lhd P_2 \equiv_T Q_1 \lhd Q_2$.

Properties (i)–(iv) descend immediately from the homomorphic definition of $*$, $\cup$, $\cap$ and $\lhd$. For example, for any interpretation I: $T(P_1 \cup P_2)(I) = T(P_1)(I) \cup T(P_2)(I) = T(Q_1)(I) \cup T(Q_2)(I) = T(Q_1 \cup Q_2)(I)$, since $P_1 \equiv_T Q_1$ and $P_2 \equiv_T Q_2$. □

The former proposition implies that if two programs are $T(P)$ equivalent then they are also indistinguishable, that is, they exhibit the same behaviours in all possible contexts. We now show that the converse holds as well, that is, programs which are indistinguishable w.r.t. $(Ob, \{*, \cup, \cap, \lhd\})$ are also $T(P)$ equivalent. Indeed we show

that a stronger result holds, namely that programs which are indistinguishable w.r.t. $(Ob, \{\cap, \lhd\})$ are also $T(P)$ equivalent. The proof of the latter result exploits a general property of immediate consequences that will be also used in the proof of other propositions. Each immediate consequence of a program P from an interpretation I is also an immediate consequence of P from a finite subset of I. This property can be naturally generalised from programs to program expressions as stated by the following lemma.

Lemma 1. *Let $P \in Pr$. For any interpretation I:*

$$A \in T(P)(I) \Rightarrow (\exists F: F \subseteq I \wedge F \text{ is finite} \wedge A \in T(P)(F)).$$

Proof. The proof is by induction on the structure of P.

P plain program: By definition of $T(P)$: $A \in T(P)(I) \Leftrightarrow (\exists \bar{B}: A \leftarrow \bar{B} \in ground(P) \wedge \bar{B} \subseteq I)$. Immediate since $\bar{B}$ is finite and since $A \in T(\bar{B})$.

$P = Q^*$: Immediate since $\forall I: T(Q^*)(I) = T(Q^*)(\emptyset)$, by definition of $*$.

$P = Q \cup R$: By definition of $\cup$: $A \in T(Q \cup R)(I) \Leftrightarrow A \in T(Q)(I) \vee A \in T(R)(I)$. By inductive hypothesis on Q and R this implies that $\exists F: F \subseteq I \wedge F$ is finite $\wedge (A \in T(Q)(F) \vee A \in T(R)F))$. By definition of $\cup$, this implies that $\exists F: F \subseteq I \wedge F$ is finite $\wedge A \in T(Q \cup R)(F)$.

$P = Q \cap R$: By definition of $\cap$: $A \in T(Q \cap R)(I) \Leftrightarrow A \in T(Q)(I) \wedge A \in T(R)(I)$. By inductive hypothesis on Q and R this implies that $(\exists F': F' \subseteq I \wedge F'$ is finite $\wedge A \in T(Q)(F')) \wedge (\exists F'': F'' \subseteq I \wedge F''$ is finite $\wedge A \in T(R)(F''))$. Let $F = F' \cup F''$, then by monotonicity of $T(Q)$ and $T(R)$ we have that $\exists F: F \subseteq I \wedge F$ finite $\wedge A \in T(Q)(F) \wedge A \in T(R)(F)$. By definition of $\cap$ this implies that $\exists F: F \subseteq I \wedge F$ is finite $\wedge A \in T(Q \cap R)(F)$.

$P = Q \lhd R$: By definition of $\lhd$: $A \in T(Q \lhd R)(I) \Leftrightarrow A \in T(Q)(I \cup T^{\omega}(R))$. By inductive hypothesis on Q, this implies that $\exists F: F \subseteq (I \cup T^{\omega}(R)) \wedge F$ is finite $\wedge A \in T(Q)(F)$. By partitioning F into two parts we have that $\exists F', F'': F' \subseteq I \wedge F'' \subseteq T^{\omega}(R) \wedge (F' \cup F'')$ is finite $\wedge A \in T(Q)(F' \cup F'')$. By monotonicity of $T(Q)$ this implies that $\exists F': F' \subseteq I \wedge F'$ is finite $\wedge A \in T(Q)(F' \cup T^{\omega}(R))$ and by definition of $\lhd$ we have that $\exists F: F \subseteq I \wedge F$ is finite $\wedge A \in T(Q \lhd R)(F)$. □

Proposition 3. *$\equiv_T$ is fully abstract with respect to $(Ob, \{\cap, \lhd\})$.*

Proof. We have to prove that $\forall P, Q \in Pr: P \cong_{\{\cap, \lhd\}} Q \Rightarrow P \equiv_T Q$. We show that if $P \not\equiv_T Q$ then $P \not\cong_{\{\cap, \lhd\}} Q$, that is if P and Q are not subsumption-equivalent then there is a context in which P and Q exibit different behaviours. By definition of $\equiv_T$, we observe that $P \not\equiv_T Q \Leftrightarrow (\exists I: T(P)(I) \not\equiv T(Q)(I))$. This means that $\exists I, A: (A \in T(P)(I) \wedge A \notin T(Q)(I)) \vee (A \notin T(P)(I) \wedge A \in T(Q)(I))$. Suppose that the first disjunct holds (the other case is analogous). Then, by Lemma 1 and by monotonicity of $T(Q)$, we have that $\exists F, A: F$ is finite $\wedge A \in T(P)(F) \wedge A \notin T(Q)(F)$. Consider now the programs $R = \{A \leftarrow\}$ and $S = \{B \leftarrow \mid B \in F\}$. We can construct a context in which

the substitution of P by Q changes the external behaviour of the context. Let $M = (P \cap R) \lhd S$ and $N = (Q \cap R) \lhd S$. Then $Ob(M) \neq Ob(N)$ since $A \in Ob(M)$ while $A \notin Ob(N)$. We observe that by definition of $\lhd$: $T(M)(\emptyset) = T((P \cap R) \lhd S)(\emptyset) = T(P \cap R)(T^{\omega}(S))$. Since $T^{\omega}(S) = F$ and by definition of $\cap$, we have that $T(M)(\emptyset) = T(P)(F) \cap T(R)(F)$. Since $T(R)(F) = \{A\}$ and $A \in T(P)(F)$, we conclude that $T(M)(\emptyset) = \{A\}$ and hence, by continuity of $T(M)$, $A \in T^{\omega}(M)$. On the other hand, $T(N)(\emptyset) = T((Q \cap R) \lhd S)(\emptyset) = T(Q \cap R)(T^{\omega}(S)) = T(Q \cap R)(F) = T(Q)(F) \cap T(R)(F)$. Since $A \notin T(P)(F)$ and $T(R)(F) = \{A\}$, we have that $T(N)(\emptyset) = \emptyset$ and hence $A \notin T^{\omega}(N)$. □

Proposition 3 implies that the equivalence relation $\equiv_T$ is also fully abstract with respect to $(Ob, \{*, \cup, \cap, \lhd\})$. This is due to the following general observation.

Observation 1. For any *Pr*, *Ob*, *com*, let $\equiv$ be an equivalence relation over *Pr*. If $\equiv$ is the fully abstract compositional equivalence relation for (Ob, Com) then $\equiv$ is the fully abstract equivalence relation for any (Ob, Com') such that $Com' \supset Com$, provided that $\equiv$ is a congruence for Com'.

On the basis of this observation, it is easy to see that subsumption-equivalence is the fully abstract compositional equivalence not only for $(Ob, \{\cap, \lhd\})$, but also for any (Ob, Com') such that

$$\{\cap, \lhd\} \subseteq Com' \subseteq \{*, \cup, \cap, \lhd\}.$$

Corollary 1. $\equiv_T$ *is fully abstract for* $(Ob, \{*, \cup, \cap, \lhd\})$.

Proposition 2 and Corollary 1 state that subsumption-equivalence is *the* fully abstract compositional equivalence relation for $(Ob, \{*, \cup, \cap, \lhd\})$. This means that the $T(P)$ semantics induces the coarsest equivalence relation on programs w.r.t. $(Ob, \{*, \cup, \cap, \lhd\})$, in that any other denotation of programs one may choose must induce the same equivalence relation to be compositional and fully abstract.

Going back to the example of natural numbers discussed in the introduction, we can now conclude that the $T(P)$ semantics correctly distinguishes the programs:

P	Q
$nat(zero) \leftarrow$	$nat(s(zero)) \leftarrow$
$nat(s(x)) \leftarrow nat(x)$	$nat(zero) \leftarrow$
	$nat(s(x)) \leftarrow nat\,(x)$

Corollary 1 states that indistinguishable programs are $T(P)$ equivalent and therefore $P \not\equiv_T Q \Rightarrow P \not\cong_{\{*, \cup, \cap, \lhd\}} Q$. Following the proof of Proposition 3, we see that if we consider the program:

R
$nat(s(zero)) \leftarrow$

then we can construct a context in which the two programs P and Q exhibit different observable behaviours, namely $Ob(P \cap R) \neq Ob(Q \cap R)$.

5. Relations with other semantics

In the previous section, we have shown that the equivalence relation induced by the $T(P)$ semantics (subsumption-equivalence) is both compositional and fully abstract with respect to the set of compositions $\{*, \cup, \cap, \triangleleft\}$. In this section, we relate subsumption-equivalence to other well known equivalence relations induced by different semantics for logic programs. More precisely, we show that equivalence relations coarser than subsumption-equivalence are both compositional and fully abstract for proper subsets of the set of operations of the algebra.

We start from the results presented by Maher [17], where various formulations of equivalence for logic programs are studied and compared. The equivalence relation induced by the $T(P)$ semantics is one of these equivalences. As already discussed in the previous section, the syntactic notion of subsumption-equivalence gives an equivalent formulation of the same equivalence relation. Another equivalence relation considered in [17] is induced by a refinement of the $T(P)$ semantics, defined by means of a $T(P) + Id$ function. As in the previous case, an equivalent formulation is given in terms of a syntactic notion of *weak subsumption-equivalence*. Furthermore, logical equivalence ($\models P \leftrightarrow Q$) and the corresponding equivalence when only Herbrand models are considered ($HM(P) = HM(Q)$) are studied. It is shown that these two equivalent relations can be equivalently formulated in terms of the functional semantics defined in [15]. Finally, the standard equivalence relation induced by the operational semantics of logic programs is considered, which identifies programs with same success set ($SS(P) = SS(Q)$) and the latter coincides with the least Herbrand model semantics [22].

Different formulations of equivalence are also compared in terms of their relative strength. In addition to the previously mentioned correspondences, it is shown that subsumption-equivalence is strictly finer than weak subsumption-equivalence, which is in turn strictly finer than logical equivalence, which is turn strictly finer than operational equivalence. Recall that an equivalence relation $\equiv_1$ is finer than another equivalence relation $\equiv_2$ ($\equiv_1 \subseteq \equiv_2$) if and only if whenever $P \equiv_1 Q$ then $P \equiv_2 Q$. Furthermore $\equiv_1$ is strictly finer than $\equiv_2$ ($\equiv_1 \subset \equiv_2$) if and only if $\equiv_1$ is finer than $\equiv_2$ and $\equiv_2$ is not finer than $\equiv_1$.

Some of the results presented in [17] are summarised in Fig. 1, where an arrow from $\equiv_1$ to $\equiv_2$ denotes that $\equiv_1$ is strictly finer than $\equiv_2$ (viz. $\equiv_1 \subset \equiv_2$). Other notions of equivalence, concerning program completion and finite failure, and studied in [17].

It is easy to observe that subsumption-equivalence is a congruence for any set of compositions $Com \subset \{*, \cup, \cap, \triangleleft\}$. On the other hand, as the number of composition operations decreases, the set of possible contexts shrinks. Therefore, programs which

$$
\begin{array}{ccc}
T(P) = T(Q) & \longleftrightarrow & P \text{ s-e } Q \\
\downarrow & & \\
T(P) + Id = T(Q) + Id & \longleftrightarrow & P \text{ w s-e } Q \\
\downarrow & & \\
\models P \leftrightarrow Q & \longleftrightarrow & HM(P) = HM(Q) \\
\downarrow & & \\
SS(P) = SS(Q) & \longleftrightarrow & LHM(P) = LHM(Q)
\end{array}
$$

Fig. 1.

are distinguishable under $(Ob, \{*, \cup, \cap, \lhd\})$ may be indistinguishable under (Ob, Com), for some $Com \subset \{*, \cup, \cap, \lhd\}$.

In the next sections, we show that the chain of equivalence relations composed by weak subsumption-equivalence, logical equivalence and least Herbrand model equivalence coincides with the chain of fully abstract compositional equivalence relations for proper subsets of the operations of the algebra, obtained by dropping one operation at a time from the set of compositions.

5.1. *Weak subsumption-equivalence*

Let us first consider the set of compositions $Com = \{*, \cup, \cap\}$, obtained by dropping $\lhd$ from the set of operations of the algebra. Subsumption-equivalence is a congruence for $\{*, \cup, \cap, \lhd\}$, and it is therefore a congruence also for the smaller set of compositions $\{*, \cup, \cap\}$. On the other hand, subsumption-equivalence is not fully abstract for this set of operations. Indeed, there are programs which are not $T(P)$ equivalent, though they cannot be distinguished operationally by means of $*$, $\cup$ and $\cap$. For instance, it can be shown that the programs:

P	Q
$r \leftarrow$	$r \leftarrow$
	$s \leftarrow s$

are indistinguishable under $(Ob, \{*, \cup, \cap\})$ though they are not $T(P)$ equivalent.

In order to obtain the fully abstract equivalence relation for $(Ob, \{*, \cup, \cap\})$, subsumption-equivalence must be slightly weakened so to identify some programs which are distinguished by the $T(P)$ semantics. It turns out that the $T(P)$ semantics is too fine for $(Ob, \{*, \cup, \cap\})$ just because it takes into account tautologies which possibly occur in a program. Recall that a tautology is a logic formula which always evaluates to *true*, and therefore a definite clause is a tautology if an exact copy of the head appears in the body of a clause. For instance, $s \leftarrow s$ and $s(x) \leftarrow r(x), s(x)$ are tautologies.

Let us consider the *weak subsumption-equivalence* relation introduced by Maher [17]. Two programs are weakly subsumption-equivalent if and only if the two

programs without tautologies are subsumption-equivalent. Weak subsumption-equivalence can be characterised in terms of the function $T(P)$ as follows.

Definition 2. *Let P be a logic program. Then for any interpretation I:*

$$(T(P) + Id)(I) = T(P)(I) \cup I.$$

Proposition 4 (Maher [17]). *P is weakly subsumption-equivalent to $Q \Leftrightarrow (T(P) + Id = T(Q) + Id)$.*

Let us name $\equiv_{T+Id}$ the equivalence relation induced by $T(P) + Id$, that is,

$$(P \equiv_{T+Id} Q) \Leftrightarrow (T(P) + Id = T(Q) + Id).$$

We first observe that $\equiv_{T+Id}$ is a congruence for $(Ob, \{*, \cup, \cap\})$.

Proposition 5. *$\equiv_{T+Id}$ is a congruence for $(Ob, \{*, \cup, \cap\})$.*

Proof. (1) $\equiv_{T+Id}$ preserves Ob: Immediate since $(T(P) + Id = T(Q) + Id) \Rightarrow (Ob(P) = Ob(Q))$ (see Fig. 1).

(2) $\equiv_{T+Id}$ is a congruence for $\{*, \cup, \cap\}$: we consider the following cases.

(i) $\forall P, Q \in Pr$: $(P \equiv_{T+Id} Q \Rightarrow P^* \equiv_{T+Id} Q^*)$: By Definition 2 and by definition of $*$, $\forall I$: $(T(P^*) + Id)(I) = T(P^*)(I) \cup I = T^\omega(P) \cup I$. Since $P \equiv_{T+Id} Q$ and $(T(P) + Id = T(Q) + Id) \Rightarrow (Ob(P) = Ob(Q))$ we have that $T^\omega(P) = T^\omega(Q)$ and hence $\forall I$: $(T(P^*) + Id)(I) = T^\omega(P) \cup I = T^\omega(Q) \cup I = (T(Q^*) + Id)(I)$.

(ii) $\forall P_1, P_2, Q_1, Q_2 \in Pr$: $((P_1 \equiv_{T+Id} Q_1 \wedge P_2 \equiv_{T+Id} Q_2) \Rightarrow P_1 \cup P_2 \equiv_{T+Id} Q_1 \cup Q_2)$: By Definition 2 and by definition of $\cup$, $\forall I$: $(T(P_1 \cup P_2) + Id)(I) = T(P_1 \cup P_2)(I) \cup I = T(P_1)(I) \cup T(P_2)(I) \cup I$. Since $P_1 \equiv_{T+Id} Q_1$ and $P_2 \equiv_{T+Id} Q_2$, we have that $T(P_1)(I) \cup T(P_2)(I) \cup I = T(Q_1)(I) \cup T(Q_2)(I) \cup I$ and hence that $\forall I$: $(T(P_1 \cup P_2) + Id)(I) = (T(Q_1 \cup Q_2) + Id)(I)$.

(iii) $\forall P_1, P_2, Q_1, Q_2 \in Pr$: $((P_1 \equiv_{T+Id} Q_1 \wedge P_2 \equiv_{T+Id} Q_2) \Rightarrow P_1 \cap P_2 \equiv_{T+Id} Q_1 \cap Q_2)$: Analogous to (ii). □

We now show that the weak equivalence-relation $\equiv_{T+Id}$ is also fully abstract for $(Ob, \{*, \cup, \cap\})$. As in the case of subsumption-equivalence (Section 4), we are able to prove a stronger result, namely that $\equiv_{T+Id}$ is fully abstract for $(Ob, \{\cup, \cap\})$.

Proposition 6. *$\equiv_{T+Id}$ is fully abstract for $(Ob, \{\cup, \cap\})$.*

Proof. We have to prove that $\forall P, Q \in Pr$: $P \cong_{\{\cup,\cap\}} Q \Rightarrow P \equiv_{T+Id} Q$. We show that if $P \not\equiv_{T+Id} Q$ then $P \ncong_{\{\cup,\cap\}} Q$. By definition of $\equiv_{T+Id}$, we observe that $P \not\equiv_{T+Id} Q \Leftrightarrow \exists I, A$: $((A \in T(P)(I) \cup I) \wedge (A \notin T(Q)(I) \cup I)) \vee ((A \notin T(P)(I) \cup I) \wedge (A \in T(Q)(I) \cup I))$. Suppose that the first disjunct holds (the other case is analogous).

By definition of set-theoretic union this implies that $\exists I, A$: $A \in T(P)(I) \wedge A \notin T(Q)(I) \wedge A \notin I$. Then, by Lemma 1 and by monotonicity of $T(Q)$, we have that $\exists F, A$: F is finite $\wedge A \in T(P)(F) \wedge A \notin T(Q)(F) \wedge A \notin F$. Consider now the programs $R = \{A \leftarrow\}$ and $S = \{B \leftarrow \mid B \in F\}$. We can construct a context in which the substitution of P by Q changes the external behaviour of the context. Let $M = (P \cap R) \cup S$ and $N = (Q \cap R) \cup S$. Then $Ob(M) \neq Ob(N)$ since $A \in Ob(M)$ while $A \notin Ob(N)$. We observe that by definition of $\cup$: $T(M)(\emptyset) = T((P \cap R) \cup S)(\emptyset) = T(P \cap R)(\emptyset) \cup T(S)(\emptyset)$. Since $T(S)(\emptyset) = F$, we have that $T(M)(\emptyset) \supseteq F$. Moreover, we observe that, by monotonicity of $T(M)$: $T^2(M)(\emptyset) = T(M)(T(M)(\emptyset)) \supseteq T(M)(F) = T((P \cap R) \cup S)(F)$. By definition of $\cup$ and $\cap$, we observe that $T^2(M)(\emptyset) \supseteq T(P \cap R)(F) \cup T(S)(F) \supseteq T(P \cap R)(F) = T(P)(F) \cap T(R)(F)$. Since $T(R)(F) = \{A\}$ and $A \in T(P)(F)$ we conclude that $A \in T^2(M)(\emptyset)$ and hence $A \in T^\omega(M)$. On the other hand, by definition of $\cup$ and $\cap$: $T(N)(\emptyset) = T((Q \cap R) \cup S)(\emptyset) = T((Q \cap R))(\emptyset) \cup T(S)(\emptyset) = (T(Q)(\emptyset) \cap T(R)(\emptyset)) \cup T(S)(\emptyset)$. Since $T(R)(\emptyset) = \{A\}$ and $A \notin T(Q)(\emptyset)$, we have that $T(N)(\emptyset) = T(S)(\emptyset) = F$. Moreover $T^2(N)(\emptyset) = T(N)(T(N)(\emptyset)) = T((Q \cap R) \cup S)(F) = (T(Q)(F) \cap T(R)(F)) \cup T(S)(F)$. Since $A \notin T(Q)(F)$ we conclude that $T^2(N)(\emptyset) = F = T(N)(\emptyset)$ and hence $T^\omega(N) = F$. Therefore $A \notin Ob(N)$ since $A \notin F$. □

By virtue of Observation 1, Propositions 5 and 6 imply the full abstraction result for $\equiv_{T+Id}$ w.r.t $(Ob, \{*, \cup, \cap\})$.

Corollary 2. *$\equiv_{T+Id}$ is fully abstract for $(Ob, \{*, \cup, \cap\})$.*

Corollary 2 states that weak subsumption-equivalence is fully abstract for $(Ob, \{*, \cup, \cap\})$. As a by-product, it also proves that the two programs given at the beginning of this section:

P	Q
$r \leftarrow$	$r \leftarrow$
	$s \leftarrow s$

are indistinguishable under $(Ob, \{*, \cup, \cap\})$ since they are weakly subsumption-equivalent and since $P \equiv_{T+Id} Q \Rightarrow P \cong_{\{*, \cup, \cap\})} Q$.

Another consequence of Corollary 2 is that the weak subsumption-equivalence relation is not compositional with respect to the operation $\lhd$. To illustrate this fact, consider again the program P and Q above, along with the program:

R
$s \leftarrow$

and observe that $P \equiv_{T+Id} Q$ while $P \lhd R \not\equiv_{T+Id} Q \lhd R$.

5.2. Logical equivalence

Let us restrict further the set of compositions $\{*, \cup, \cap\}$ and consider the set of compositions $Com = \{*, \cup\}$. Weak subsumption-equivalence is a congruence for $\{*, \cup, \cap\}$, and it is therefore a congruence also for the smaller set of compositions $\{*, \cup\}$. On the other hand, weak subsumption-equivalence is not fully abstract for this set of operations. Indeed, there are programs which are not weakly subsumption–equivalent, though they cannot be distinguished operationally by means of $*$ and $\cup$. For instance, it can be shown that the programs:

P	Q
$r \leftarrow$	$r \leftarrow s$
$s \leftarrow$	$s \leftarrow$

are indistinguishable under $(Ob, \{*, \cup\})$ though they are not weakly subsumption-equivalent.

Let us consider logical equivalence, which in the case of logic programs coincides with Herbrand models equivalence. Two programs are equivalent if and only if they have the same set of Herbrand models ($HM(P) = HM(Q)$). Recall that a Herbrand interpretation I is a Herbrand model for a program P if and only if $T(P)(I) \subseteq I$ ([1]). The equivalence relation $\equiv_{HM}$ on Pr is defined as follows.

Definition 3. Let $P, Q \in Pr$, then

$$P \equiv_{HM} Q \Leftrightarrow \forall I: (T(P)(I) \subseteq I \Leftrightarrow T(Q)(I) \subseteq I).$$

The equivalence relation $\equiv_{HM}$ considers two programs equivalent if and only if they have the same set of Herbrand models. Notice that the programs P and Q of the previous example are identified by $\equiv_{HM}$.

The equivalence relation $\equiv_{HM}$ is a congruence for $\{\cup\}$. Indeed the Herbrand models of the union of two programs can be determined by the Herbrand models of the separate programs. Actually, given two prorams P and Q, a Herbrand interpretation I is a model for $P \cup Q$ if and only if I is a model for both P and Q. It is easy to show that the equivalence relation $\equiv_{HM}$ is a congruence also for $\{*\}$, and hence the following proposition holds.

Proposition 7. *$\equiv_{HM}$ is a congruence for $(Ob, \{*, \cup\})$.*

Proof. (1) $\equiv_{HM}$ preserves Ob: Immediate since $HM(P) = HM(Q) \Rightarrow Ob(P) = Ob(Q)$ (see Fig. 1).

(2) $\equiv_{HM}$ is a congruence for $\{*, \cup\}$: We consider the following cases.

(i) $\forall P, Q \in Pr$: $(P \equiv_{HM} Q \Rightarrow P^* \equiv_{HM} Q^*)$: By definition of $*$, $\forall I$: $T(P^*)(I) = T^{\omega}(P)$. Since $HM(P) = HM(Q) \Rightarrow Ob(P) = Ob(Q)$, we have that $\forall I$: $T(P^*)(I) = T(Q^*)(I)$ and hence $\forall I$: $(T(P^*)(I) \subseteq I) \Leftrightarrow (T(Q^*)(I) \subseteq I)$.

(ii) $\forall P_1, P_2, Q_1, Q_2 \in Pr$: $((P_1 \equiv_{HM} Q_1 \wedge P_2 \equiv_{HM} Q_2) \Rightarrow P_1 \cup P_2 \equiv_{HM} Q_1 \cup Q_2)$: By definition of $\cup$, $\forall I$: $T(P_1 \cup P_2)(I) \subseteq I \Leftrightarrow (T(P_1)(I) \cup T(P_2)(I)) \subseteq I \Leftrightarrow (T(P_1)(I) \subseteq I) \wedge (T(P_2)(I) \subseteq I)$. Since $P_1 \equiv_{HM} Q_1$ and $P_2 \equiv_{HM} Q_2$, we have that $\forall I$: $T(P_1 \cup P_2)(I) \subseteq I \Leftrightarrow T(Q_1 \cup Q_2)(I) \subseteq I$. □

We now show that if two programs cannot be distinguished by means of the compositions $\cup$ and $*$ then they have the same Herbrand models. As in the previous sections, we prove a stronger result, namely that $\equiv_{HM}$ is fully abstract w.r.t. $(Ob, \{\cup\})$.

Proposition 8. *$\equiv_{HM}$ is fully abstract with respect to $(Ob, \{\cup\})$.*

Proof. We have to prove that for any $P, Q \in Pr$: $P \cong_{\{\cup\}} Q \Rightarrow P \equiv_{HM} Q$. We show that if $P \not\equiv_{HM} Q$ then $P \not\cong_{\{\cup\}} Q$. By definition of $\equiv_{HM}$: $P \not\equiv_{HM} Q \Leftrightarrow (\exists I$: $T(P)(I) \subseteq I \wedge T(Q)(I) \not\subseteq I) \vee (\exists I$: $T(P)(I) \not\subseteq I \wedge T(Q)(I) \subseteq I)$. Suppose that the first disjunct holds (the other case is analogous). If $\exists I$: $T(Q)(I) \not\subseteq I$ then $\exists I, A$: $A \in T(Q)(I) \wedge A \notin I$. By Lemma 1 and by monotonicity of $T(Q)$, this implies that $\exists I, F, A$: $F \subseteq I \wedge F$ is finite $\wedge A \in T(Q)(F) \wedge A \notin F$. We can now construct a context in which the substitution of P by Q changes the external behaviour of the context. Consider the program $R = \{B \leftarrow \mid B \in F\}$ and let $M = P \cup R$ and $N = Q \cup R$. Then $A \notin Ob(M)$ while $A \in Ob(N)$. We first show that $T^{\omega}(P \cup R) \subseteq I$. We prove, by induction on n, that $\forall n$: $T^n(P \cup R) \subseteq I$. The base case ($n = 0$) is trivial since $T^0(P \cup R) = \emptyset$. Assume now that $T^n(P \cup R) \subseteq I$. By definition of powers of T: $T^{n+1}(P \cup R) = T(P \cup R)(T^n(P \cup R))$. By the inductive hypothesis and by monotonicity of $T(P \cup R)$ we have that $T^{n+1}(P \cup R) \subseteq T(P \cup R)(I)$. By definition of $\cup$, since $F \subseteq I$ and since $T(P)(I) \subseteq I$, we observe that $T(P \cup R)(I) = T(P)(I) \cup T(R)(I) \subseteq I$. Hence $T^{n+1}(P \cup R) \subseteq I$. Therefore, by continuity of $T(P \cup R)$, we have that $T^{\omega}(P \cup R) \subseteq I$ and since $A \notin I$ we conclude that $A \notin T^{\omega}(P \cup R)$. On the other hand, $A \in Ob(N)$ since $\{A\} \subseteq T(Q)(F) = T(Q)(T(R)(\emptyset))$. By definition of $\cup$ and monotonicity of $T(Q)$, $T(Q)(T(R)(\emptyset)) \subseteq T(Q)(T(Q \cup R)(\emptyset)) \subseteq T(Q \cup R)(T(Q \cup R)(\emptyset))$. Therefore $\{A\} \subseteq T(Q \cup R)(T(Q \cup R)(\emptyset)) = T^2(Q \cup R) \subseteq T^{\omega}(Q \cup R)$, by continuity of $T(Q \cup R)$. □

By Observation 1, Propositions 7 and 8 imply that logical equivalence is fully abstract w.r.t. $(Ob, \{*, \cup\})$.

Corollary 3. *$\equiv_{HM}$ is fully abstract with respect to $Ob, \{*, \cup\})$.*

Corollary 3 also proves that the two programs given at the beginning of this section:

P	Q
$r \leftarrow$	$r \leftarrow s$
$s \leftarrow$	$s \leftarrow$

are indistinguishable under $(Ob, \{*, \cup\})$ since they are logically equivalent and since $P \equiv_{HM} Q \Rightarrow P \cong_{\{*, \cup\}} Q$.

Another consequence of Corollary 3 is that logical equivalence is not a congruence for the other two operations $\cap$ and $\lhd$ of the algebra. To illustrate this fact, consider again the programs P and Q above, along with the program:

R
$r \leftarrow$

and observe that $P \equiv_{HM} Q$ while $P \cap R \not\equiv_{HM} Q \cap R$. Similar considerations apply to the case of $\lhd$.

5.3. Least Herbrand model semantics

Let us finally restrict the set of compositions $\{*, \cup\}$ to the set $Com = \{*\}$. Actually, $Com = \{*\}$ corresponds to the case in which no composition of programs is allowed. In fact, the unary operation $*$ supports the encapsulation rather than the composition of separate programs.

Logical equivalence is a congruence for $\{*, \cup\}$, and it is therefore a congruence also for the smaller set of compositions $\{*\}$. On the other hand, logical equivalence is not fully abstract for this set of operations. Indeed, there are programs which are not logically equivalent, though they cannot be distinguished operationally by means of $*$ only. For instance, it is easy to see that the empty program and the program

Q
$r \leftarrow s$

are indistinguishable under $(Ob, \{*\})$ though they are not logically equivalent.

Intuitively, two programs are indistinguishable under $(Ob, \{*\})$ if and only if they have the same observable behaviour. Let us consider the standard least Herbrand model semantics, where a program is denoted by its least Herbrand model or, equivalently, by its success set [22]. The equivalence relation $\equiv_{LHM}$ on Pr is defined as follows.

Definition 4. *Let $P, Q \in Pr$, then*

$$P \equiv_{LHM} Q \Leftrightarrow T^{\omega}(P) = T^{\omega}(Q).$$

Such a denotation corresponds to the case where the observables alone determine the equivalence relation on programs. It is easy to show that $\equiv_{LHM}$ is the fully abstract compositional equivalence for $(Ob, \{*\})$.

Proposition 9. *$\equiv_{LHM}$ is the fully abstract compositional equivalence relation for $(Ob, \{*\})$.*

Proof. (1) $\equiv_{LHM}$ is a congruence for $(Ob, \{*\})$. We consider the following.

(1.1) $\equiv_{LHM}$ preserves Ob: Immediate, by definition of Ob.

(1.2) $\equiv_{LHM}$ is a congruence for $\{*\}$: We have to prove that $\forall P, Q \in Pr$: $P \equiv_{LHM} Q \Rightarrow P^* \equiv_{LHM} Q^*$. Obvious, since, by definition of $*$, $T^\omega(P^*) = T^\omega(P)$.

(2) $\equiv_{LHM}$ is fully abstract with respect to $(Ob, \{*\})$: We have to prove that for any $P, Q \in Pr$: $P \cong_{\{*\}} Q \Rightarrow P \equiv_{LHM} Q$. Obvious since if $P \not\equiv_{LHM} Q$ then $Ob(P) \neq Ob(Q)$. □

Notice that it is easy to show that the equivalence relation $\equiv_{LHM}$ is is too coarse to model other program composition operations. For instance, $\equiv_{LHM}$ is not a congruence for $\cup$, in that the least Herbrand model of a program cannot be determined by the least Herbrand models of its clauses. The same consideration applies both to $\cap$ and $\lhd$.

6. Discussion

6.1. *The chain of fully abstract equivalence relations*

In Section 4 we have shown that subsumption-equivalence ($\equiv_T$) is both compositional and fully abstract for the whole set of operations $\{*, \cup, \cap, \lhd\}$ of the algebra (Corollary 1). Furthermore, in Section 5, we have shown that the chain of equivalence relations composed by weak subsumption-equivalence ($\equiv_{T+Id}$), logical equivalence ($\equiv_{HM}$) and least Herbrand model equivalence ($\equiv_{LHM}$) coincides with the chain of fully abstract compositional equivalence relations for the subsets of the operations of the algebra $\{*, \cup, \cap\}$, $\{*, \cup\}$ and $\{*\}$, respectively (Corollaries 2 and 3, Proposition 9). These results are summarised in Fig. 2, where *C* stands for compositional, **FAC** for fully abstract and compositional, and – for non-compositional.

As shown in [17] subsumption-equivalence, weak subsumption-equivalence, logical equivalence and least Herbrand model equivalence form a totally ordered set of equivalence relations, that is:

$$\equiv_T \subset \equiv_{T+Id} \subset \equiv_{HM} \subset \equiv_{LHM}.$$

When considering different equivalence relations induced by different semantics for programs, an equivalence relation is coarser than another one if the corresponding

	$\{*, \cup, \cap, \lhd\}$	$\{*, \cup, \cap\}$	$\{*, \cup\}$	$\{*\}$
$\equiv_T$	**FAC**	*C*	*C*	*C*
$\equiv_{T+Id}$	—	**FAC**	*C*	*C*
$\equiv_{HM}$	—	—	**FAC**	*C*
$\equiv_{LHM}$	—	—	—	**FAC**

Fig. 2.

semantics *abstracts more* from the syntax of programs. In our case, weak subsumption-equivalence abstracts more from the syntax of programs than subsumption-equivalence, and so on.

We now discuss how the chain of fully abstract equivalence relations relates the operations $*$, $\cup$, $\cap$ and $\lhd$ to the structural features of programs. Let us start from the following observation.

Observation 2. Let $\equiv_1$ and $\equiv_2$ be two equivalence relations such that $\equiv_1 \subset \equiv_2$. If $\equiv_1$ is the fully abstract equivalence relation for $(Ob, Com \cup \{c\})$ and $\equiv_2$ is the fully abstract equivalence relation for (Ob, Com) then:

$$\forall P, Q: \quad (P \not\equiv_1 Q \ \wedge \ P \equiv_2 Q) \ \Leftrightarrow \ P \text{ and } Q \text{ can be distinguished only via } c.$$

Indeed by definition of full abstraction: $(P \not\equiv_1 Q \ \wedge \ P \equiv_2 Q) \Leftrightarrow (P \not\cong_{Com \cup \{c\}} Q \ \wedge \ P \cong_{Com} Q)$.

Let us now consider the chain of fully abstract equivalence relations discussed so far. The above observation states, for instance, that programs which are weakly subsumption-equivalent but not subsumption-equivalent can be distinguished only via the operation $\lhd$.

Let us consider the first step of the chain, that is moving from subsumption-equivalence to weak subsumption-equivalence. Actually, weak subsumption-equivalence weakens subsumption-equivalence by identifying those programs which are distinguished by the latter because of tautologies. By Observation 2, this means that programs which are $T(P)$-equivalent up to tautologies cannot be distinguished operationally without the operation $\lhd$. From a programming perspective, tautologies can be employed to realise a form of implementation hiding. Intuitively speaking, tautologies may play the role of *bridge rules* linking the definition of a predicate to its actual implementation. Consider the following program Q implementing the operations of insertion and deletion on multisets:

Q

$insert(e, s, [e \mid s]) \leftarrow$

$delete(e, [\,], [\,]) \leftarrow$

$delete(e, [e \mid s], s) \leftarrow$

$delete(e, [f \mid s], [f \mid news]) \leftarrow e \neq f, delete(e, s, news)$

The actual implementation of multisets (by lists) as well as of the operations on multisets (by operations on lists) can be hidden to the other programs by importing the program Q from the program:

P

$insert(x, y, z) \leftarrow insert(x, y, z)$

$delete(x, y, z) \leftarrow delete(x, y, z)$

In the composition $P \lhd Q$, the tautological clauses of P support the import of the implementation of the operations *insert* and *delete*, contained in Q, by virtue of the definition of $\lhd$, that is: $T(P \lhd Q)(I) = T(P)(I \cup T^{\omega}(Q))$. It is worth observing that P is not subsumption-equivalent, for instance, to the empty program in that the two programs can be operationally distinguished by importing the program Q. On the other hand, P and the empty program are weakly subsumption-equivalent in that they are operationally indistinguishable without the operation $\lhd$. The example illustrates that the actual usefulness of tautologies for programming does rely on the exploitation of the composition operation $\lhd$. Indeed, without the import operation $\lhd$, tautologies lose their role of bridge rules and can therefore be ignored.

Let us now consider the second step of the chain, that is moving from weak subsumption-equivalence to logical equivalence. Intuitively speaking, logical equivalence identifies all the programs which have the same set of Herbrand models, disregarding the way and the number of inference steps necessary to derive an atomic formula. For instance, the programs

$r \leftarrow s$ $\quad$ $r \leftarrow$
$s \leftarrow$ $\quad$ $s \leftarrow$

are not weakly subsumption-equivalent, in that in the former r can be derived only through s, while in the latter r can be derived independently of s. By Observation 2, programs which are logically equivalent though not weakly subsumption-equivalent can be distinguished only by using the operation $\cap$. From a programming perspective, the operation $\cap$ takes into account the way in which atomic formulae are derived in the programs. Indeed, $\cap$ enforces a step-by-step co-operation between programs that have to agree at each derivation step. Consider for instance, the programs:

P	Q
wobbly_wheel $\leftarrow$	*wobbly_wheel* $\leftarrow$ *flat_tyre*
flat_tyre $\leftarrow$	*flat_tyre* $\leftarrow$

which are not weakly subsumption-equivalent, though logically equivalent. In Q the fault *wobbly_wheel* does depend on the fault *flat_tyre*, while in P there is no causal dependency between the two faults. The programming effect of this difference emerges when the two programs are composed with other programs by means of the operation $\cap$. For instance, the program:

R
wobbly_wheel $\leftarrow$
flat_tyre $\leftarrow$ *wobbly_wheel*

permits to operationally distinguish P from Q, since $Ob(P \cap R) = \{$*wobbly_wheel*, *flat_tyre*$\}$ while $Ob(Q \cap R) = \emptyset$.

Let us finally consider the last step of the chain, that is moving from logical equivalence to least Herbrand model equivalence. The least Herbrand model of a program contains only the atomic formulae which are provable in the program,

while any other information concerning possible compositions with other programs is lost. On the other hand, non-least Herbrand models contain also formulae which are not provable in the program alone, but which may become provable after adding further clauses to the program. For instance, the program:

P
$path(x, y) \leftarrow arc(x, y)$
$path(x, y) \leftarrow arc(x, z), path(z, y)$

is not logically equivalent to the empty program, while they have the same least Herbrand model. Again, by Observation 2, these programs can be distinguished only by using the operation $\cup$. For instance, the program:

Q
$arc(a, b) \leftarrow$
$arc(b, c) \leftarrow$

allows to distinguish P from the empty program when applying the operation $\cup$.

6.2. *Other sets of compositions*

We have considered several subsets of the set of compositions $\{*, \cup, \cap, \lhd\}$ of the algebra, and related them to well known equivalence relations. From a programming perspective, however, it is interesting to consider also other subsets of the set of composition operations. For instance the set consisting only of the operations $\cup$ and $\lhd$ is particularly interesting in the context of modular logic programming, as discussed in [8].

Let us first introduce the notion of *justified interpretation*, inspired by the notion of admissible interpretation proposed in [4].

Definition 5. Let $P \in Pr$ and let I, H be Herbrand interpretations. Then

I is an interpretation for P justified by $H \Leftrightarrow I = V^{\omega}(P, H)$,

where $V^{\omega}(P, H)$ is the limit of following sequence:

$V^0(P, H) = \emptyset$,

$V^{n+1}(P, H) = T(P)(H \cup V^n(P, H))$.

Intuitively speaking, I is the limit of $T^i(P)(\emptyset)$ where at each step $T(P)$ possibly uses a given set of hypotheses H. It is easy to observe that justified interpretations are also models of a program.

Proposition 10. *I is an interpretation for P justified by $H \Rightarrow T(P)(I) \subseteq I$.*

We now consider the equivalence relation induced by justified models.

Definition 6. Let $P, Q \in Pr$, then

$$P \equiv_J Q \Leftrightarrow \forall H: (V^\omega(P,H) = V^\omega(Q,H)).$$

Namely, two programs P and Q are J-equivalent if and only if they have the same justified models, that is, for any Herbrand interpretation H, I is model for P justified by H if and only if I is a model for Q justified by H.

It is interesting to see how the equivalence relation $\equiv_J$ relates to the other equivalence relations that have been studied in the previous sections. The next proposition shows that $\equiv_J$ is coarser than subsumption-equivalence, and finer than logical equivalence.

Proposition 11. $\equiv_T \subset \equiv_J \subset \equiv_{HM}$.

Proof. We first show that $\equiv_T \subseteq \equiv_J$. Let $P, Q \in Pr$ such that $P \equiv_T Q$. Then for any interpretation H: $V^\omega(P,H) = V^\omega(Q,H)$ since $\forall I$: $T(P)(I) = T(Q)(I)$. We now show that $\equiv_J \subseteq \equiv_{HM}$. Let us first observe that $\forall I: (T(P)(I) \subseteq I \Leftrightarrow V^\omega(P,I) \subseteq I)$. Suppose now that $P \not\equiv_{HM} Q$. Then, by definition of $\equiv_{HM}$: $(\exists I: T(P)(I) \subseteq I \wedge T(Q)(I) \not\subseteq I) \vee (\exists I: T(P)(I) \not\subseteq I \wedge T(Q)(I) \subseteq I)$. Suppose that the first disjunct holds (the other case in analogous). Then, by the above observation: $\exists I$: $V^\omega(P,I) \subseteq I \wedge V^\omega(Q,I) \not\subseteq I$ and hence $P \not\equiv_J Q$.

Finally, to show that $\equiv_T \neq \equiv_J$ and that $\equiv_J \neq \equiv_{HM}$ consider the following programs:

P_1	Q_1	P_2	Q_2
$r \leftarrow$	$r \leftarrow s$	$r \leftarrow$	$r \leftarrow$
$s \leftarrow$	$s \leftarrow$		$s \leftarrow s$

It is easy to observe that $P_1 \not\equiv_T Q_1$ while $P_1 \equiv_J Q_1$, and that $P_2 \not\equiv_J Q_2$ while $P_2 \equiv_{HM} Q_2$ □

Moreover, it is worth noting that the equivalence relation $\equiv_J$ is not comparable with weak subsumption-equivalence. Indeed, in the previous proof, we can also observe that $P_1 \not\equiv_{T+Id} Q_1$ and that $P_2 \equiv_{T+Id} Q_2$. The relation between $\equiv_J$ and the other equivalence relations is summarised in Fig. 3.

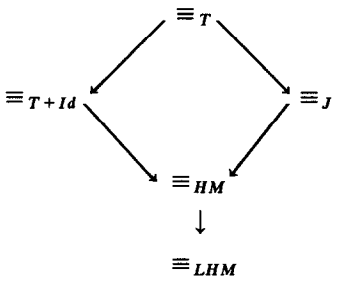

Fig. 3

We now show that the equivalence relation $\equiv_J$ is a congruence for the operation $*$, $\cup$ and $\lhd$.

Proposition 12. *$\equiv_J$ is a congruence for $(Ob, \{*, \cup, \lhd\})$.*

Proof. (1) $\equiv_J$ preserves Ob: Immediate since, by Proposition 11, $P \equiv_J Q \Rightarrow Ob(P) = Ob(Q)$.

(2) $\equiv_J$ is a congruence for $\{*, \cup, \lhd\}$. We consider the following cases.

(i) $\forall P, Q \in Pr: (P \equiv_J Q \Rightarrow P^* \equiv_J Q^*)$: By definition of $*$, $\forall H: V^\omega(P^*, H) = T^\omega(P)$. Since $P \equiv_J Q \Rightarrow Ob(P) = Ob(Q)$, we have that $\forall H: V^\omega(P^*, H) = T^\omega(P) = T^\omega(Q) = V^\omega(Q^*, H)$.

(ii) $\forall P_1, P_2, Q_1, Q_2 \in Pr: ((P_1 \equiv_J Q_1 \wedge P_2 \equiv_J Q_2) \Rightarrow P_1 \cup P_2 \equiv_J Q_1 \cup Q_2)$: The justified models of the union of two program expressions can be determined by the justified models of the separate program expressions, since $\forall H$:

$$V^\omega(P_1 \cup P_2, H) = \min\{M \mid I \subseteq (M \cup H) \Rightarrow (V^\omega(P_1, I) \subseteq M \wedge V^\omega(P_2, I) \subseteq M)\}.$$

Therefore, since $P_1 \equiv_J Q_1 \wedge P_2 \equiv_J Q_2$, we have that $\forall H: V^\omega(P_1 \cup P_2, H) = V^\omega(Q_1 \cup Q_2, H)$.

(iii) $\forall P_1, P_2, Q_1, Q_2 \in Pr: ((P_1 \equiv_J Q_1 \wedge P_2 \equiv_J Q_2) \Rightarrow P_1 \lhd P_2 \equiv_J Q_1 \lhd Q_2)$: We first show that $\forall H: V^\omega(P_1 \lhd P_2, H) = V^\omega(P_1, H \cup T^\omega(P_2))$. We prove, by induction on n, that $\forall H, n: V^n(P_1 \lhd P_2, H) = V^n(P_1, H \cup T^\omega(P_2))$. The base case ($n = 0$) is trivial since $\forall H: V^0(P_1 \lhd P_2, H) = V^0(P_1, H \cup T^\omega(P_2)) = \emptyset$. Assume now that $V^n(P_1 \lhd P_2, H) = V^n(P_1, H \cup T^\omega(P_2))$. Then $V^{n+1}(P_1 \lhd P_2, H) = T(P_1 \lhd P_2)(H \cup V^n(P_1 \lhd P_2, H))$ and, by definition of $\lhd$, $V^{n+1}(P_1 \lhd P_2, H) = T(P_1)(H \cup T^\omega(P_2) \cup V^n(P_1 \lhd P_2, H))$. By inductive hypothesis and by definition of V^{n+1}, we have that $V^{n+1}(P_1 \lhd P_2, H) = V^{n+1}(P_1, H \cup T^\omega(P_2))$. Now, since by Proposition 11 $P_2 \equiv_J Q_2 \Rightarrow T^\omega(P_2) = T^\omega(Q_2)$, we also have that $\forall H: V^\omega(P_1 \lhd P_2, H) = V^\omega(P_1, H \cup T^\omega(Q_2))$. Finally, since $P_1 \equiv_J Q_1$ we conclude that $\forall H: V^\omega(P_1 \lhd P_2, H) = V^\omega(Q_1, H \cup T^\omega(Q_2)) = V^\omega(Q_1 \lhd Q_2, H)$. □

We now show that $\equiv_J$ is the fully abstract equivalence relation for $(Ob, \{*, \cup, \lhd\})$. Actually we first prove a stronger result, namely that $\equiv_J$ is fully abstract w.r.t. $(ob, \{\lhd\})$.

Lemma 2. *Let $P, Q \in Pr$. Then $\forall H$:*

$$V^\omega(P, H) \neq V^\omega(Q, H) \Rightarrow (\exists F: V^\omega(P, F) \neq V^\omega(Q, F) \wedge F \text{ is finite}).$$

Proof. We observe that $V^\omega(P, H) \neq V^\omega(Q, H) \Leftrightarrow \exists A: (A \in V^\omega(P, H) \wedge A \notin V^\omega(Q, H)) \vee (A \notin V^\omega(P, H) \wedge A \in V^\omega(Q, H))$. Suppose that the first disjunct holds (the other case

is analogous). We first show, by induction on n, that $\forall n$: $A \in V^n(P,H) \Rightarrow (\exists F_n$: $A \in V^n(P,F_n) \wedge F_n \subseteq H \wedge F_n$ is in finite). The base case is trivial. We then observe that: $A \in V^{n+1}(P,H) \Leftrightarrow A \in T(P)(H \cup V^n(P,H))$ and hence, by Lemma 1, $\exists K$: $A \in T(P)(K) \wedge K \subseteq (H \cup V^n(P,H)) \wedge K$ is finite. Let $K = K_H \cup K_V$ such that $K_H \subseteq H$ and $K_V \subseteq V^n(P,H)$, then by inductive hypothesis we have that $\exists K, F_n$: $A \in T(P)(K) \wedge K = K_H \cup K_V \wedge K_H \subseteq H \wedge K_V \subseteq V^n(P,F_n) \wedge F_n \subseteq H \wedge F_n$ is finite. Then, since $T(P)$ is monotonic, we have that $A \in T(P)(K)$ implies that $A \in T(P)$ $(F_n \cup K_H \cup V^n(P,F_n))$, and since $V^n(P,F_n) \subseteq V^n(P,F_n \cup K_H)$ this implies that $A \in V^{n+1}(P,F_n \cup K_H)$. Therefore if $A \in V^{n+1}(P,H)$ then $\exists F_{n+1}$: $A \in V^{n+1}(P,F_{n+1}) \wedge$ $F_{n+1} \subseteq H \wedge F_{n+1}$ is finite).

Hence $\exists A$: $(A \in V^\omega(P,H) \wedge A \notin V^\omega(Q,H))$ implies that $\exists A,F$: $(A \in V^\omega(P,F) \wedge F \subseteq$ $H \wedge F$ is finite $\wedge A \notin V^\omega(Q,H))$ and hence, since $V^\omega(Q,F) \subseteq V^\omega(Q,H)$ by definition of V, $\exists A,F$: $(A \in V^\omega(P,F) \wedge F$ is finite $\wedge A \notin V^\omega(Q,F))$ which implies that $\exists F$: $(V^\omega(P,F) \neq V^\omega(Q,F) \wedge F$ is finite). □

Proposition 13. *$\equiv_J$ is fully abstract for $(Ob, \{\lhd\})$.*

Proof. We have to prove that $\forall P,Q \in Pr$: $P \cong_{\{\lhd\}} Q \Rightarrow P \equiv_J Q$. We show that if $P \not\equiv_J Q$ then $P \not\cong_{\{\lhd\}} Q$. By definition of $\equiv_J$: $P \not\equiv_J Q \Leftrightarrow (\exists H$: $V^\omega(P,H) \neq V^\omega(Q,H))$. By Lemma 2, this implies that $\exists F$: $V^\omega(P,F) \neq V^\omega(Q,F) \wedge F$ is finite. Let $R = \{A \leftarrow \mid A \in F\}$. We first show that $V^\omega(P,F) = T^\omega(P \lhd R)$. We prove, by induction on n, that $\forall n$: $V^n(P,F) = T^n(P \lhd R)$. The base case $(n = 0)$ is trivial since $V^0(P,F) = T^0(P \lhd R) = \emptyset$. Assume now that $V^n(P,F) = T^n(P \lhd R)$. Then by definition of V^{n+1} and by inductive hypothesis we have that $V^{n+1}(P,F) = T(P)(F \cup$ $V^n(P,F)) = T(P)(F \cup T^n(P \lhd R))$. Since $T^\omega(R) = F$ we also have that $V^{n+1}(P,F) =$ $T(P)(T^\omega(R) \cup T^n(P \lhd R)) = T^{n+1}(P \lhd R)$. Therefore $P \not\equiv_J Q \Rightarrow (\exists R$: $T^\omega(P \lhd R) \neq$ $T^\omega(Q \lhd R))$ and hence $\exists R$: $Ob(P \lhd R) \neq Ob(Q \lhd R))$. □

By observation 1, Propositions 12 and 13 imply that $\equiv_J$ is fully abstract for any (Ob, Com) such that

$$\{\lhd\} \subseteq Com \subseteq \{*, \cup, \lhd\}.$$

Corollary 4. *$\equiv_J$ is fully abstract for $(Ob, \{*, \cup, \lhd\})$.*

A consequence of Corollary 4 is that $\equiv_J$ is not a congruence for the operation $\cap$. To illustrate this fact, consider again the programs:

P	Q	R
$r \leftarrow$	$r \leftarrow s$	$r \leftarrow$
$s \leftarrow$	$s \leftarrow$	

and observe that $P \equiv_J Q$ while $P \cap R \not\equiv_J Q \cap R$.

The equivalence relation $\equiv_J$ concludes our study of fully abstract equivalence relations for different subsets of the set of compositions $\{*, \cup, \cap, \triangleleft\}$. Actually we have exhibited fully abstract equivalence relations for all possible subsets of $\{*, \cup, \cap, \triangleleft\}$ but $\{\cap\}$ and $\{*, \cap\}$. However the scarce programming interest of these two sets makes the search of a fully abstract equivalence relation for them a merely theoretical exercise.

6.3. Other issues

In this paper we have explored the space with logic program compositions and logic program semantics as dimensions, and we have used the notions of compositionality and full abstraction as a yard-stick to measure which set of compositions is well supported by which semantics. We wish to point out that we have fixed at least three more potentially variable dimensions a priori: The definition of observables, the notion of vocabulary, and the set of admissible programs. Indeed we have taken the standard notion of success set defined in [1, 16, 22] as the observable behaviour of a logic program. Other definitions of observable behaviour for a logic program may be considered. For instance, one may choose to observe the set of (possibly non-ground) atoms provable in a program, as done for instance in [11, 13]. Then, we have considered programs over a fixed vocabulary, defined by a set of predicate symbols and a set of function symbols. More sophisticated notions of vocabulary are studied in [13, 17]. Finally, larger classes of logic programs may be considered, in particular programs containing some form of negation. The task does not appear to be easy, especially because the choice of the proper set of observable may still be considered an open problem.

It is our opinion that working with an algebra of logic programs, where different program composition operations can be used, enriches the usability of logic programming for software construction and knowledge representation. In this light it is necessary to extend tools and techniques for the analysis and the transformation of logic programs to take into account the operations which can be applied to them. For example, it is important to design static analysis techniques such that their results on different programs can be composed together when the programs are composed via an operation. Some results both on composing separate analyses [10] and on composing separate program termination proofs [2] have already been obtained, when union is considered as the only program composition operation. It is obvious that the choice of the semantics is of fundamental importance in the design of the analysis technique. We have shown that the choice of the appropriate semantics depends on the set of operations which are considered. We are confident that the results presented here can be useful in designing proper analysis and transformation techniques in the context of an algebra of logic programs.

Acknowledgements

We would like to thank the anonymous referees for their pertinent comments and suggestions. This work has been partly supported by the Esprit BRA 6810 *Compulog 2*

and by Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo of C.N.R. under grant no. 92.01564.PF69.

References

[1] K.R. Apt, Logic programming, in: J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, Vol. B* (Elsevier, Amsterdam, 1990) 493–574.
[2] K.R. Apt and D. Pedreschi, Reasoning about termination of pure Prolog programs, *Inform. and Comput.* **106** (1993) 109–157.
[3] A. Brogi, Program construction in computational logic, Ph.D. Thesis, University of Pisa, March 1993.
[4] A. Brogi, E. Lamma and P. Mello, Compositional model-theoretic semantics for logic programs, *New Generation Computing* **11**(1) (1992) 1–21.
[5] A. Brogi, P. Mancarella, D. Pedreschi and F. Turini, Composition operators for logic theories, in: J.W. Lloyd, ed., *Computational Logic, Symp. Proc.* (Springer, Berlin, 1990) 117–134.
[6] A. Brogi, P. Mancarella, D. Pedreschi and F. Turini, Hierarchies through basic metalevel operators, in: M. Bruynooghe, ed., *Proc. 2nd Workshop on Meta-programming in Logic* (1990) 381–396.
[7] A. Brogi, P. Mancarella, D. Pedreschi and F. Turini, Theory construction in computational logic, in: J.-M. Jacquet, ed., *Constructing Logic Programs* (Wiley, New York, 1993) 241–250.
[8] A. Brogi, P. Mancarella, D. Pedreschi and F. Turini, Modular logic programming, *ACM Trans. Programming Languages Systems* **16**(4) (1994) 1361–1398.
[9] A. Brogi and F. Turini, Metalogic for knowledge representation, in J.A. Allen, R. Fikes and E. Sandewall, eds., *Principles of Knowledge Representation and Reasoning: Proc. 2nd Internat. Conf.* (Morgan Kaufmann, Los Altos, CA, 1990) 100–106.
[10] M. Codish, S.K. Debray and R. Giacobazzi, Compositional analysis of modular logic programs, in: *Proc. 20th Ann. ACM Symp. on Principles of Programming Languages* (ACM Press, New York, 1993) 451–464.
[11] M. Falaschi, G. Levi, M. Martelli and C. Palamidessi, Declarative modeling of the operational behavior of logic languages, *Theoret. Comput. Sci.* **69**(3) (1989) 289–318.
[12] M. Fitting, Enumeration operators and modular logic programming *J. Logic Programming* **4** (1987) 11–21.
[13] H. Gaifman and E. Shapiro, Fully abstract compositional semantics for logic programs, in: *Proc. 16th ACM Symp. on Principles of Programming Languages* (ACM Press, New York, 1989) 134–142.
[14] G. Gottlob and A. Leitsch, On the efficiency of subsumption algorithms, *J. ACM* **32**(2) 1985) 280–295.
[15] J.L. Lassez and M.J. Maher, Closures and fairness in the semantics of logic programming *Theoret. Comput. Sci.* **29** (1984) 167–184.
[16] J.W. Lloyd, *Foundations of Logic Programming* (Springer, Berlin, 2nd ed., 1987).
[17] M.J. Maher, Equivalences of logic programs, in: J. Minker ed., *Deductive Databases and Logic Programming* (Morgan Kaufmann, Los Altos, CA, 1988) 627–658.
[18] P. Mancarella and D. Pedreschi, An algebra of logic programs, in: R.A. Kowalski and K.A. Bowen eds., *Proc. 5th Internat. Conf. on Logic Programming.* (MIT Press, Mascow, 1988) 1006–1023.
[19] R. Milner, Fully abstract models for typed λ-calculi, *Theoret. Comput. Sci.* **4** (1977) 1–23.
[20] R. O'Keefe, Towards an algebra for constructing logic programs, in: J. Cohen and J. Conery, eds., *Proc. IEEE Symp. on Logic Programming* (IEEE Computer Society Press, Silver Spring, MD, 1985) 152–160.
[21] G.D. Plotkin, LCF considered as a programming language, *Theoret. Comput. Sci.* **5** (1977) 223–256.
[22] M.H. van Emden and R.A. Kowalski, The semantics of predicate logic as a programming language, *J. ACM* **23**(4) (1976) 733–742.