Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

Differential logic programs: semantics and programming methodologies

A. Bossi, M. Bugliesi, M. Gabbrielli, G. Levi, M.C. Meo

Computer Science/Department of Software Technology

CS-R9363 1993

Differential Logic Programs: Semantics and Programming Methodologies

A. Bossi M. Bugliesi
Dipartimento di Matematica Pura ed Applicata
Università di Padova, Via Belzoni 7, 35131 Padova, Italy
{*isa@zenone,michele@goedel*}*.unipd.it*

M. Gabbrielli
CWI
P.O. Box 94079, 1090 GB Amsterdam, The Netherlands
gabbri@cwi.nl

G. Levi M. C. Meo
Dipartimento di Informatica
Università di Pisa, Corso Italia 40, 56100 Pisa, Italy
{*levi,meo*}*@di.unipi.it*

Abstract

The purpose of this paper is to make a contribution towards the integration of Object-Orientation and logic programming. We introduce the notion of *differential* programs, logic programs annotated to make their external interfaces explicit. Similarly to classes in the O-O paradigm, differential programs can be organized in *isa* hierarchies. The *isa*-composition of differential programs captures the semantics of several mechanisms such as static and dynamic *overriding* inheritance as well as a form of *extension* inheritance defined in term of composition by union of clauses. The application of the programming discipline we propose is illustrated on a concrete programming example.

We give a proof-theoretic semantics for *isa*-hierarchies: we define an inference system which extends SLD resolution to take into account the inheritance mechanisms encompassed by the *isa* operator. Then we introduce a corresponding operator which provides a syntactic counterpart of the *isa*-composition. The new operator, denoted by $\lhd$, transforms any *isa* hierarchy *HP* into an equivalent "flat" program $HP_\lhd$ whose proof theoretic semantics is defined in terms of the standard notion of SLD derivation. Finally we define a fixed point semantics which is $\lhd$-compositional and which models correctly the answer substitutions of programs. By virtue of the aforesaid correspondence between *isa*-hierarchies and $\lhd$-composite programs, we obtain a compositional semantics for *isa* hierarchies. The semantics of differential programs generalizes previous work on OR-compositional semantics for logic programs. It is obtained by resorting to the notion of *context-sensitive* interpretations, an extension of the Ω-denotations of [3] defined as sets of non-ground clauses.

1991 Mathematics Subject Classification: 68N17, 68Q55, 68T15.

CR Categories: D.3.1, D.3.3, F.3.2, I.2.3.

Keywords and Phrases: logic programming, object-orientation, inheritance, compositional semantics.

Note: A preliminary version of this paper appeared in [2].

1 Inheritance in logic languages

The power of Horn clause logic as a programming language was pointed out for the first time in [17] and since then it has gained the interest of a still growing research community. The most appealing features of logic as a programming language rely both in the elegance of its semantic characterization and in the declarativity of its computational model. As best summarized by Zaniolo in [33], the rule based reasoning of logic, combined with adequate tools for efficiently storing and retrieving large amounts of information could provide a realistic basis for the development of efficient knowledge base systems. As a matter of fact, its use in the development of knowledge base applications has promptly disclosed one major weakness of Horn Clause Logic as a programming language. In fact, in spite of its declarativity, logic programming turns out not to scale very well when it comes to designing practical applications. Its unit of abstraction – relations – appears to be too fine grained to support the development and the maintenance of large programs.

This need for a more structured approach to software development has motivated a wide research effort in the logic programming community during the last decade. Inspired by the experience gained in related fields, several approaches have been taken and different solutions have been proposed in the recent literature. One of the currently most promising directions in this area is based on the idea of integrating into a logical framework some of the distinguishing notions of the Object-Oriented programming paradigm: *abstraction* and *inheritance*.

From a logical point of view, an object – the O-O unit of abstraction – has a natural interpretation as a logic theory: an object is simply a collection of axioms which describe what is true about the object itself. Under this assumption, the design of a coherent semantic model for a logic language extended to incorporate the notion of inheritance can be attempted at different levels. At the operational level, it amounts to defining a new inference system which combines this mechanism with the deductive process of resolution. At the declarative level, it rises two interesting issues: firstly, the problem of characterizing inheritance in terms of the standard notions of satisfiability and truth found in classical logic; secondly, the problem of capturing the compositional properties inherent in the incremental approach to software development entailed by inheritance.

Inheritance. Our view of inheritance conforms with the one nowadays widely accepted in the Object-Oriented community. An intuitive justification for this interpretation has been proposed by Cook in [10]. Inheritance is viewed as a mechanism for differential programming, i.e. a mechanism for constructing new program components by specifying how they differ from the existing ones. Differential programming is achieved by using filters to modify the external behaviour of existing components. Accordingly, a modified version of a component is obtained by defining a new component that performs some special operations and possibly calls the original one. This idea is

illustrated by the following example.

Example 1.1 *Consider the following definitions:*

CLASS *student*
$\quad whoAmI = print(\text{“a}Student'')$
$\quad whoAreYou = \text{SELF} : whoAmI$

CLASS *cs-student* IS_A *student*
$\quad whoAmI = print(\text{“a}CsStudent'')$

$aStudent$:= NEW *student*
$anotherStudent$:= NEW *cs-student*

We have two classes, student *and* cs-student*, and two corresponding instances. Class* cs-student *is a subclass of* student *and redefines one of its superclass' methods. The invocation* NEW class *returns an instance of* class *whereas the expression* object : message *denotes the request for* object *to execute the* method *associated with* message*. We are interested in the answers to the two following message-sents.*

(a) $aStudent : WhoAreYou.$
(b) $anotherStudent : WhoAreYou.$

The result of evaluating (a) is straightforward. The message is sent to aStudent *and the answer is: “a* Student*”.*

Case (b) is more interesting: the result depends on what the self-reference SELF *refers to. We have two choices and two corresponding answers. The first is to interpret* SELF *as the object in which the self-reference occurs:* aStudent*. The corresponding answer, exemplified in figure 1, shows that the modification* csStudent *only partially affects the external behaviour of the the original component* students*. In [10] Cook*

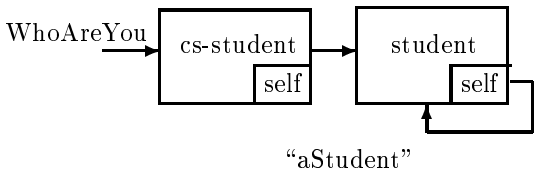

Figure 1: Static interpretation of Self

argues (indeed convincingly) that what we actually expect here is that SELF *refers to the composite object obtained by applying the modification to the original component. This characterization, which constitutes the main motivation to Cook's approach, is obtained by the interpretation illustrated in figure 2.*

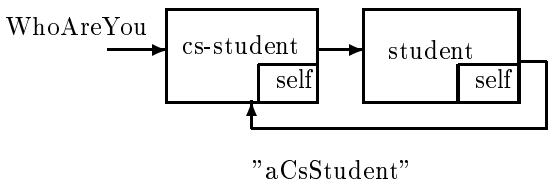

Figure 2: Dynamic Interpretation of Self

Figure 2 also provides a justification for inheritance as a mechanism for deriving modified versions of recursive structures.

In an independent study [29], Reddy adopts a similar approach but distinguishes different forms of inheritance: the interpretation given by figure 2 is classified as *dynamic inheritance* – a là Smalltalk [16] – as opposed to the *static* mechanism exhibited by languages like *Simula-67* and depicted in figure 1. The approach we take in this paper follows Reddy's classification and extends it to account for a notion of *extension* (to be contrasted with *overriding*) inheritance.

Compositionality. The adequacy of a semantic characterization for a language is typically measured on the account of how effective the semantics is for defining the meaning for programs written in the language. However, in view of a modular approach to program development, the semantics of a language should actually aim at characterizing the meaning of program fragments rather than of stand-alone programs. This is in fact crucial to be able to define the meaning of a composite program on the account of the meaning of its components. A semantics with these properties is said to be *homomorphic* or *compositional*. More precisely, we say that the semantics (or invariant) $[\![\cdot]\!]$ is compositional with respect to a composition operation $\circ$ ($\circ$-compositional for short) if, given two program components A and B, the relation $[\![A \circ B]\!] = [\![A]\!]\ \sigma(\circ)\ [\![B]\!]$ holds for a suitable choice of the *homomorphism* σ which maps the syntactic operator $\circ$ onto the corresponding semantic operator $\sigma(\circ)$.

Compositionality is also a fundamental property for reasoning about component equivalence. Denote with $\approx_{Ob}$ the relation of computational equivalence associated with the notion of observable Ob. If the semantics $[\![\cdot]\!]$ is $\approx_{Ob}$-correct (i.e. it identifies *only* $\approx_{Ob}$-equivalent components) and it is $\circ$-compositional, then it can be used to justify the replacement of equivalent components in any $\circ$-composite context. In fact, when $[\![\cdot]\!]$ is $\approx_{Ob}$-correct, for any two components A and B, $[\![A]\!] = [\![B]\!] \Rightarrow A \approx_{Ob} B$. If the semantics is also $\circ$-compositional, then

$$[\![A]\!] \;=\; [\![B]\!] \;\Longrightarrow\; (C_1 \circ \cdots \circ A \circ \cdots \circ C_n) \approx_{Ob} (C_1 \circ \cdots \circ B \circ \cdots \circ C_n)$$

for any choice of components $C_1, \ldots, C_n$. Any semantics satisfying the above implication is said to be $(Ob, \circ)$-congruent.

In the context of a hierarchical composition of program based on inheritance, a natural interpretation of compositionality is the following. If $P_1, \ldots, P_n$ are program components and *isa* is the specialization operator, then the composition P_n *isa* $\cdots P_2$ *isa* P_1 should be read as the right-associative composition P_n *isa* $(\cdots(P_2$ *isa* $P_1)\cdots)$. Under this assumption, compositionality can then be expressed by imposing the following, weaker, condition on the homomorphism σ and the semantics $[\![\cdot]\!]$:

$$[\![P_n \ isa \ \cdots P_2 \ isa \ P_1]\!] = [\![P_n]\!] \ \sigma(isa) \ (\cdots [\![P_3]\!] \ \sigma(isa) \ ([\![P_2]\!] \ \sigma(isa) \ [\![P_1]\!])\cdots)$$

This notion of compositionality will provide the basis for our discussion throughout the paper.

Goals and Outline. The purpose of this paper is to make a contribution towards the integration of Object-Orientation and logic programming. The approach we follow is based on the notion of *differential* programs, logic programs with explicit annotations qualifying three classes of the *exportable* predicates. These programs can be organized in *isa* hierarchies where each program inherits the definitions of the exportable predicates contained in the program's ancestors in the hierarchy. The purpose of the annotations is twofold. On one hand, the introduction of exportable predicates accounts for standard forms of encapsulation and hiding; on the other, it allows different forms of inheritance to coexist in the language.

In fact, the *isa*-composition of differential programs captures the semantics of several mechanisms such as static and dynamic *overriding* inheritance as well as a form of *extension* inheritance defined in term of composition by union of clauses. The evaluation of a goal in a hierarchy of differential programs is defined proof-theoretically in terms of an extension of SLD resolution.

We also discuss an equivalent semantics, based on standard SLD resolution. This is obtained by defining a new composition operator, denoted by $\lhd$, which provides the syntactic counterpart of the *isa* operator. In fact we show that any *isa*-hierarchy can be transformed by means of $\lhd$ into a single "flat" program, whose SLD-semantics is equivalent to the extended SLD-semantics of the *isa*-hierarchy.

We then introduce a declarative semantics for differential programs following the approach described in [13]. In that paper, the idea is that a declarative characterization of the programs' operational behaviour can be obtained by accommodating in the programs' interpretations more complex syntactic objects. The notion of observable which the semantics of [13] is meant to capture is given by the computed answer substitutions (cas in the following) of a program. Here, we generalize the notion of interpretations based on clauses which was adopted in [3] to obtain a (cas, OR)-congruent semantics and we introduce the notion of *context-sensitive* interpretations to obtain a $(cas, \lhd)$-congruent semantics.

By virtue of the aforesaid correspondence between *isa* hierarchies and $\lhd$ programs, this semantics is a $\approx_{cas}$-correct and *isa*-compositional declarative semantics for the proof-procedure defined for *isa* hierarchies of differential programs.

The rest of the paper is organized as follows. In section 2 we define the notion of differential programs and the proof-theoretic semantics for *isa* hierarchies of differential programs. Then we define the composition operator $\lhd$ and we show the related equivalence result. In section 3, we define the declarative semantics of differential programs and show its compositional and correctness properties. In section 4 we discuss the application of the differential approach on a concrete example. Finally in section 5 we discuss the relation of our approach with the existing literature in the field. To enhance readability some technical lemmas and their proofs are relegated to a final appendix.

2 Differential Logic Programs

A differential program is a program P annotated by three sets of *exported* predicate symbols:

Σ: statically inherited predicates (Σ-*predicates*);

Δ: dynamically inherited predicates (Δ-*predicates*);

Θ: extensible predicates (Θ-*predicates*).

We assume the three sets are mutually disjoint, and their union is contained in the set $\pi(P)$ of the predicate symbols occurring in P. The remaining predicates, $\pi(P) \setminus (\Sigma \cup \Delta \cup \Theta)$ will be henceforth referred to as *internal* predicates and denoted with $\iota(P)$. We assume that the symbols for internal predicates range over an alphabet $\Im$ which is disjoint from the alphabets used for Σ, Δ and Θ predicates. For any program we also denote by $\kappa(P)$ the set of the predicates defined in P (p is defined in P if there exists a clause in P whose head's predicate symbol is p) and we define as *open* the predicates in the set $\Omega(P) = (\Sigma \setminus \kappa(P)) \cup \Delta \cup \Theta$. The qualification *open* is used here to emphasize the fact that the definition for these predicates can be modified by composing P with other programs. This is not the case for internal predicates and, as explained below, for static predicates which are locally defined in P.

Statically and dynamically inherited predicates are evaluated according to an overriding semantics. The distinction between the two sets Σ and Δ reflects the distinction between two different forms of inheritance we would like to coexist. The idea is the following: a program P is to be understood as part of a structured context of the form C *isa* P *isa* D and the evaluation of a goal depends on the annotation of the goal's predicate symbols. A Σ-predicate is evaluated in P using P's local definition or any definition inherited from the context D. The local definition, if there is any, overrides the inherited one. Hence, any occurrence in P of a goal for a static predicate which is also defined in P, is bound to the local definition independently of the context in which P occurs. Conversely, the evaluation of a Δ-predicate in P uses the local definition or the inherited one, only if no definition for the same predicate name is provided by the context C. If C does provide a definition, than this definition overrides in P the local or inherited one.

The annotation Θ models a different composition mechanism defined with an *extension* semantics: the definition of a Θ-predicate in P can be *extended* (to be contrasted

with overridden) with the definition found in C and/or in D.

The following example illustrates the use of these composition mechanisms.

Example 2.1 *Consider again the two classes of example 1.1, defined now as differential logic programs and extended with new methods.*

CS_Student	*isa*	**Student**
$whoAmI('aCS\_Student').$		$whoAmI('aStudent').$
		$whoAreYou(X) :- whoAmI(X).$
$address('CS\_Dept').$		$address('univ\_hall').$
		$adm\_addr(X) :- address(X).$
		$course(X) :- required(X).$
$required('LogicProg').$		$required('4thLevel').$
.....		....

The use of different annotations for the exportable predicates of the two programs is motivated by the behaviour we expect in response to the different queries for the hierarchy **CS_Student** *isa* **Student**. *Consider first the query :-whoAreYou(X). Here, the expected answer is* $X =$ *'aCS_Student' and it can be obtained by taking* whoAmI *to be a* Δ*-predicate. To see this, note that* **CS_Student** *inherits the definition for* whoAreYou(X) *from* **Student** *and, being* whoAmI *a* Δ*-predicate, the evaluation of the call* whoAmI(X) *uses the definition contained in* **CS_Student**. *The evaluation of the goal :-*address(X) *follows the same pattern as long as* address *is a* Σ *or* Δ *predicate. What's more interesting is the query :-*adm_addr(X). *Here, the expected answer is* $X =$ univ_hall *for we assume that the administrative address of a student is independent of the department where that student belongs. This behaviour can be modeled by defining* address *to be a* Σ*-predicate: this guarantees that the evaluation of the call* address(X) *uses the definition local to* **Student**.

Finally, we can model the fact that a **CS_Student** *is expected to take all of the courses required for a* **Student** *by defining* course *and* required *to be* Θ *predicates.*

2.1 isa-hierarchies of Programs

We can make this intuitive picture precise by formally defining the rules for evaluating a goal in a generic *isa*-hierarchy of differential programs. Let *HP* be the hierarchy P_n *isa* P_{n-1} *isa* $\cdots$ *isa* P_1. It follows from the previous discussion that the evaluation of a goal in *HP* is well defined if and only if every predicate symbol occurring in the P_is has a unique identity in terms of membership to the corresponding annotations Σ_is, Δ_is and Θ_is. The following condition ensures this property.

Definition 2.2 (Compatibility) *Let* $\langle \Sigma_P, \Delta_P, \Theta_P \rangle$*-*$P$ *and* $\langle \Sigma_Q, \Delta_Q, \Theta_Q \rangle$*-*$Q$ *be two differential programs.* P *and* Q *are said to be compatible provided that the following*

condition holds:

$$\begin{aligned}\pi(P) \cap \pi(Q) &= (\Sigma_P \cap \Sigma_Q) \cup (\Theta_P \cap \Theta_Q) \\ &\quad \cup (\Delta_P \cap \Delta_Q) \cup (\iota(P) \cap \iota(Q)).\end{aligned}$$

We will henceforth consider only hierarchies of compatible programs, that is we assume that in P_n isa P_{n-1} isa $\cdots$ isa P_1 all component P_is are pairwise compatible. Moreover, we consider only the evaluation of goals whose predicate symbols are exported by at least one of the component programs. Hence, the internal predicates are thought of as encapsulated within the components and hence their definition is not exported by the hierarchies. The evaluation rules are given below in Natural Deduction style extending those reported in [7]. The notation $HP \vdash_\theta G$ should be read "G succeeds in HP with substitution θ".

If G is a conjunctive goal, $G = G_1, G_2$ then evaluating G in HP amounts to evaluating in HP each of the conjuncts Namely:

$$\frac{HP \vdash_\theta G_1 \quad HP \vdash_\sigma G_2\theta}{HP \vdash_{\theta\sigma} G_1, G_2}$$

If G is an atomic goal, say $p(t)$, then the first step consists of selecting in the hierarchy HP a component which contains a clause for p. Formally:

$$\frac{P_k, HP \vdash_\sigma p(t)}{HP \vdash_\sigma p(t)}$$

The annotation for the predicate p determines the different ways of selecting the component P_k: the top-most component of HP for Σ and Δ-predicates; any one for Θ-predicates. Formally:

$$\begin{array}{ll}(1)\ p \in \bigcup_i(\Sigma_{P_i} \cup \Delta_{P_i}) & \Rightarrow k = max\{j \mid p \in \kappa(P_j)\} \\ (2)\ p \in \bigcup_i(\Theta_{P_i}) & \Rightarrow k \in \{j \mid p \in \kappa(P_j)\}\end{array}$$

The relation $P_j, HP \vdash_\theta G$ is defined similarly, the main difference being that, together with the choice of the component P_k, it encompasses the selection of a matching clause for the selected atom in the goal. The case for conjunctive goals again splits the evaluation on each of the conjuncts:

$$\frac{P_j, HP \vdash_\theta G_1 \quad P_j, HP \vdash_\sigma G_2\theta}{P_j, HP \vdash_{\theta\sigma} G_1, G_2}$$

The case for atomic goals defines the main step:

$$\frac{P_k, HP \vdash_\sigma G\theta \quad (\theta = mgu(p(t), p(\overline{t}))\ \ \&\ \ p(\overline{t}) \text{:-} G \in P_k)}{P_j, HP \vdash_{\theta\sigma} p(t)}$$

where P_k is chosen according to one of the following conditions

$$\begin{array}{lll}(1')\ p \in \Sigma_{P_j} & \& & k = max\ \{i \leq j \mid P_i \text{ defines } p\} \\ (2')\ p \in \Delta_{P_j} & \& & k = max\ \{i \leq n \mid P_i \text{ defines } p\} \\ (3')\ p \in \Theta_{P_j} & \& & k \in \{i \mid P_i \text{ defines } p\} \\ (4')\ p \in \iota(P_j) & \& & k = j\end{array}$$

The rules for the empty goal are the following

$$\frac{}{HP \vdash_\epsilon \Box} \qquad\qquad \frac{}{P_j, HP \vdash_\epsilon \Box}$$

where $\Box$ denotes the empty goal and ϵ the empty substitution).

Remarks. $(1')$ and $(2')$ (and similarly (1) and (2)) formalize the overriding semantics of Σ- and Δ-predicates. For Δ-predicates, the search for a matching clause for $p(t)$ stops at P_k, the top-most component of HP which defines p. For Σ-predicates the search ignores the components $P_n \cdots P_{j+1}$.

The operational semantics of the *isa*-composition can be now defined in terms of the above rules as follows. Call an *isa*-proof for G in HP a proof-tree rooted at $HP \vdash_\theta G$ whose internal nodes are instances of one of the above inference rules and whose leaf nodes are labeled by $HP \vdash_\epsilon \Box$ or by $P_j, HP \vdash_\epsilon \Box$. Then, the evaluation of G in HP yields the substitution θ iff there exists an *isa*-proof for $HP \vdash_\theta G$. When $HP \vdash_\theta G$, we say that the restriction of θ to the variables of G ($\theta_{|G}$) is a *computed answer* of G in HP.

Note that the proof of a conjunctive goal always selects the conjunct to be reduced according to a left-most selection rule. This involves no loss of generality since:

(i) the choice of component P_k whence the evaluation of each of the conjuncts is to start depends only on the predicate symbol of the selected conjunct (and not on the conjunct's instance). Hence,

(ii) the independence from the selection rule which holds for SLD-refutations ensures that the set of answer substitution computed by an *isa*-proof is independent of the choice of the selection rule.

2.2 Syntactic Program Composition

The operational semantics of *isa*-hierarchies of differential programs can be given equivalently in terms of SLD-resolution by introducing a composition operator, denoted by $\lhd$, which maps any *isa*-hierarchy onto a corresponding differential program. The definition of $\lhd$-composition provides also the link between the operational semantics defined in terms of *isa*-proofs and the compositional semantics for differential programs which will be introduced in section 3. As discussed in the introduction, we will assume the composition operator $\lhd$ to be right associative and thus that a hierarchical composition $P_n \lhd P_{n-1} \lhd \cdots \lhd P_1$ be interpreted as $P_n \lhd (P_{n-1} \lhd \cdots \lhd P_1)$.

Let's first introduce a little notation and terminology. For any (non atomic) goal G, $Pred(G)$ stands for the set of predicate symbols of the atoms occurring in G. We also denote with $\tilde{B}$ a conjunction of atoms, with $\tilde{X}$ a tuple of variables. $G \overset{\vartheta}{\leadsto}_{P,R} \tilde{B}$ denotes an SLD derivation in the program P from G to the resolvent $\tilde{B}$, where R is the selection rule and ϑ is the composition of the mgu's used in the derivation. Instead we

write $G \stackrel{\vartheta}{\longmapsto}_P \Box$ when G has an SLD refutation with computed answer substitution ϑ. The notion of computed answer is standard, i.e. ϑ is the restriction to the variables of G of the mgu's computed in the refutation (R is omitted because ϑ is independent of R in the case of refutations). Finally we assume the reader familiar with the standard notions of logic programming reported in [1] and [19].

The definition of $\lhd$-composition is based on the following notion of renaming.

Definition 2.3 (Renaming) *Let Π and Γ, $\Gamma \subseteq \Pi$, be two sets of predicate symbols. We denote with $\Phi_{\Gamma,\Pi}$ a family of injective functions which rename each predicate symbol in Γ with a new internal predicate symbol which does not belong to Π. If $\phi \in \Phi_{\Gamma,\Pi}$, then*

$$\phi(p) \quad = \quad \begin{cases} p_\phi \in \Im \setminus \Pi & \text{if } p \in \Gamma \\ p & \text{otherwise} \end{cases}$$

Given $\phi \in \Phi_{\Gamma,\Pi}$ and a program P such that $\pi(P) \subseteq \Pi$, we will henceforth abuse the notation and write $\phi(P)$ to denote the program obtained by applying the renaming $\phi \in \Phi_{\Gamma,\Pi}$ to all the predicate symbols occurring in P.

Definition 2.4 (Syntactic composition) *Let $\langle \Sigma_P, \Delta_P, \Theta_P \rangle$-$P$ and $\langle \Sigma_Q, \Delta_Q, \Theta_Q \rangle$-$Q$ be two differential programs. The composition $P \lhd Q$ is defined provided that the two programs are compatible in the sense of definition 2.2. If $P \lhd Q$ is defined, it denotes the differential program*

$$P \lhd Q \quad = \quad \langle \Sigma, \Delta, \Theta \rangle\text{-}(\xi(P) \cup \phi(Q^\star))$$

where ξ, ϕ and $Q^\star$ are defined respectively as

$$\begin{array}{ll} \xi \in \Phi_{\Gamma',\Pi'} & \phi \in \Phi_{\Gamma,\Pi} \\ \Gamma' = \Sigma_P \setminus (\kappa(P) \cup \kappa(Q)) & \Gamma = (\Sigma_Q \cap \kappa(P)) \cup (\Sigma_Q \setminus \kappa(Q)) \cup (\iota(P) \cap \iota(Q)) \\ \Pi' = \pi(P \cup Q) & \Pi = \pi(\xi(P)) \cup \pi(Q) \end{array}$$

$$Q^\star = \{h \text{ :- } \tilde{B} \in Q \mid Pred(h) \notin \Delta_Q \cap \kappa(P)\}$$

and the annotation $\langle \Sigma, \Delta, \Theta \rangle$ is computed according to the following definitions:

$$\begin{array}{l} \Sigma = (\Sigma_P \cup \Sigma_Q) \cap \pi(P \lhd Q) \\ \Delta = (\Delta_P \cup \Delta_Q) \cap \pi(P \lhd Q) \\ \Theta = (\Theta_P \cup \Theta_Q) \cap \pi(P \lhd Q) \end{array}$$

Some explanations are needed at this point. A first remark concerns the annotations Σ, Δ and Θ for the composition $(P \lhd Q)$. The subsetting operations as well as the renamings ξ and ϕ might modify the set of predicate symbols occurring in P and Q. Hence, the new annotation for $P \lhd Q$ is obtained by taking the union of Σ_P, Δ_P and Θ_P respectively with Σ_Q, Δ_Q and Θ_Q and then intersecting the resulting components

with the set $\pi(P \lhd Q)$. As a consequence all the new names generated by the renamings become internal for the composite program.

To explain the renaming ξ recall that our basic assumption on the *isa*- and $\lhd$-compositions is that they are right associative. Then, we may safely assume that $(P \lhd Q)$ is not going to be furtherly composed onto any hierarchy of the form $(P \lhd Q) \lhd H$. Now consider a predicate symbol $p \in \Sigma_P$. If p is not defined either in P or in Q, (i.e. $p \in \Sigma_P \setminus (\kappa(P) \cup \kappa(Q))$) then the clauses of P whose bodies contain p will never be selected by any successful *isa*-proof for the hierarchy P *isa* Q. Correspondingly, p is transformed by the renaming operation to an internal predicate which (by definition) is not visible from the context.

As for the renaming ϕ, it is performed in order to avoid name clashes for static and internal predicates. Consider first the predicates of Q. We rename the occurrences of a predicate p in Q whenever $p \in \Sigma_Q \cap \kappa(P)$. There are two reasons for this choice. On one side, since p is static in Q, any predicate call for p in Q should refer to the original definition of p local to Q also in the composition $P \lhd Q$. On the other side, the clauses which define p in Q must be distinguished from the definition of p already existing in P (since we assume an overriding semantics). The predicates in $\Sigma_Q \setminus \kappa(Q)$ are renamed for the same reason we rename in P the predicates in $\Sigma_P \setminus (\kappa(P) \cup \kappa(Q))$. In fact, if $p \in \Sigma_Q$ and p is not defined in Q, then no clause containing p can be used for a successfull isa-proof. Finally the renaming for predicates in $\iota(P) \cap \iota(Q)$ guarantees that is that no clashes arise between internal predicates of P and Q. If the composition is defined, no other clash can arise between two predicate names of P and Q. Furthermore, since the internal predicate symbols generated via renaming range over $\Im \setminus \Pi$, any new program R compatible with P and Q, will be also compatible with $P \lhd Q$. Hence, the composition operators $\lhd$ and *isa* are defined in the exact same cases.

As for the subsetting $\star$ on Q, it is performed in order to remove from Q the clauses which define predicate symbols in Δ_Q which are also defined by P. This provides the syntactic counterpart of the overriding semantics which we assume for Δ-predicates in the *isa*-composition.

Example 2.5 *Let P, Q and R be the following differential programs:*

$$P = \begin{cases} q(a). \\ r(c). \\ s(x) \text{:-} q(x). \\ t(b). \end{cases} \qquad Q = \begin{cases} h(x) \text{:-} t(x). \\ r(a) \text{:-} q(x). \\ s(x) \text{:-} r(x). \\ s(a). \end{cases} \qquad R = \begin{cases} h(a). \\ q(x) \text{:-} h(x). \\ r(b). \\ s(x) \text{:-} t(x). \end{cases}$$

where $\Sigma_P = \{q, t\}$, $\Sigma_Q = \Sigma_R = \{h, q, t\}$, $\Delta_P = \Delta_Q = \Delta_R = \{r\}$ and $\Theta_P = \Theta_Q = \Theta_R = \{s\}$. We compute the composition $P \lhd Q \lhd R$ in two steps (recall that $\lhd$ is right-associative).

$$Q \lhd R = \begin{cases} h(x) \text{:-} t_{\xi_1}(x). \\ r(a) \text{:-} q(x). \\ s(x) \text{:-} r(x). \\ s(a). \end{cases} \qquad \begin{array}{l} h_{\phi_1}(a). \\ q(x) \text{:-} h_{\phi_1}(x). \\ s(x) \text{:-} t_{\phi_1}(x). \end{array}$$

The clause $r(b) \in R$ *has been deleted because* $r \in \Delta_R \cap \kappa(Q)$. *The renamings are* $\xi_1 \in \Phi_{\{t\},\{h,q,r,s,t\}}$ *and* $\phi_1 \in \Phi_{\{h,t\},\{h,q,r,s,t,t_{\xi_1}\}}$. *The new annotation for* $Q \lhd R$ *is given by* $\Sigma = \{h, q\}$, $\Delta = \{r\}$ *and* $\Theta = \{s\}$. *Composing* P *on the* $Q \lhd R$ *yields the new program*

$$P \lhd (Q \lhd R) \;=\; \left\{ \begin{array}{llll} q(a). & & h(x)\text{:-}t_{\xi_1}(x). & h_{\phi_1}(a). \\ r(c). & & s(x)\text{:-}r(x). & q_{\phi_2}(x)\text{:-}h_{\phi_1}(x). \\ s(x)\text{:-}q(x). & & s(a). & s(x)\text{:-}t_{\phi_1}(x). \\ t(b). & & & \end{array} \right.$$

where the renaming is $\phi_2 \in \Phi_{\{q\},\{h_{\phi_1},q,r,s,t,t_{\xi_1}\}}$ *and the final annotation is* $\Sigma = \{h, q, t\}$, $\Delta = \{r\}$ *and* $\Theta = \{s\}$.

We mentioned earlier in this section that the composition operator $\lhd$ provides an alternative and equivalent characterization for the *isa*-composition. This equivalence follows from the tight correspondence existing between isa-proofs and successful SLD-derivations. We can in fact establish a one-to-one mapping between the steps of a successful SLD derivation in any $\lhd$-composition and the steps of a corresponding *isa*-proof for the corresponding *isa*-hierarchy. The proof of this result is based on the properties of the renamings employed in the construction of the syntactic composition.

Lemma 2.6 *Let* P *and* Q *be two compatible differential programs and let* $P \lhd Q = \xi(P) \cup \phi(Q^\star)$ *be their composition. Then, for any two predicate symbols* p_1 *and* p_2 *in* $\pi(P) \cup \pi(Q)$*:*

$$p_1 \neq p_2 \;\Rightarrow\; \xi(p_1) \neq \phi(p_2) \quad \textit{and} \quad \phi(p_1) \neq \xi(p_2)$$

Proof. *We show that* $p_1 \neq p_2 \;\Rightarrow\; \xi(p_1) \neq \phi(p_2)$. *The other case is symmetric. We have four possible cases:*

> $p_1 \in \Gamma'$, $p_2 \in \Gamma$. *This implies that* $\xi(p_1) \neq \phi(p_2)$ *since* $\xi(\Gamma') \cap \phi(\Gamma) = \oslash$. *In fact, from definition 2.4,* $\phi \in \Phi_{\Gamma,\xi(\pi(P))\cup\pi(Q)}$ *and hence* $\phi(\Gamma) \cap \xi(\pi(P)) = \oslash$. *Then* $\xi(\Gamma') \cap \phi(\Gamma) = \oslash$ *being* $\xi(\Gamma') \subseteq \xi(\pi(P))$.
>
> $p_1 \in \Gamma'$, $p_2 \notin \Gamma$. *Then* $\xi(p_1) \notin \pi(P) \cup \pi(Q)$ *whereas* $\phi(p_2) = p_2 \in \pi(P) \cup \pi(Q)$. *Hence* $\xi(p_1) \neq \phi(p_2)$.
>
> $p_1 \notin \Gamma'$, $p_2 \in \Gamma$. *This is symmetric to the previous case since* $\xi(p_1) = p_1$ *and* $\phi(p_2) \notin \xi(\pi(P)) \cup \pi(Q)$.
>
> $p_1 \notin \Gamma'$, $p_2 \notin \Gamma$. *In this case* $\xi(p_1) = p_1$ *and* $\phi(p_2) = p_2$. *Hence* $p_1 = p_2$ *which contradicts the hypothesis* $p_1 \neq p_2$.

■

Intuitively, what this lemma shows is that the renaming involved in the $\lhd$-composition on *two* programs is injective. Unfortunately, as shown by the following counter-example, this result does not extend to hierarchies consisting of $n > 2$ components.

Example 2.7 *Consider the following four differential programs:*

$$\begin{array}{cccccccc} P_4 & & isa & P_3 & isa & P_2 & isa & P_1 \\ r\,\texttt{:-}\,s. & & & r. & & h. & & r\,\texttt{:-}\,q. \end{array}$$

where r is a Δ-predicate, s and q are Σ-predicates and h is a Θ-predicate. In the construction of the hierarchy $P_4 \lhd P_3 \lhd P_2 \lhd P_1$, the occurrence of q in P_1 is renamed to q_{ϕ_1} when forming $P_2 \lhd P_1$. Then clause $r\,\texttt{:-}\,q_{\phi_1}$ is erased when forming $P_3 \lhd P_2 \lhd P_1$ and, at the next step, the occurrence of s in P_4 can be renamed to q_{ϕ_1}.

This is in fact a general problem: the occurrence of a renamed predicate might get erased by a subsetting applied at an intermediate step of the construction of the $\lhd$-composition, hence allowing that name to be used for renaming a different predicate occurring further up in the hierarchy.

To prove the equivalence we will therefore need the following, more accurate, characterization of the renamings employed in the construction of the $\lhd$-composite programs. Let $HP = P_n \; isa \; P_{n-1} \; isa \cdots isa \; P_1$ and $HP_\lhd = P_n \lhd P_{n-1} \lhd \cdots \lhd P_1$ be respectively the *isa* hierarchy and the corresponding $\lhd$-program. Let $\xi_2, \cdots, \xi_n$ and $\phi_1, \ldots \phi_{n-1}$ be the renamings applied in the construction of the hierarchy $HP_\lhd$ (ξ_j and ϕ_{j-1} are applied respectively to P_j and $H_j = P_{j-1} \lhd \ldots \lhd P_1$ when H_j is extended with P_j). We denote with ψ^j_{HP} the (composition of the) renaming(s) applied to P_j in the construction $HP_\lhd$. It is easy to see that

$$\psi^j_{HP} \;=\; \xi_j \circ \phi_j \circ \cdots \circ \phi_{n-1}$$

where ξ_1 is the identity renaming, $\psi^n_{HP} = \xi_n$ and $\circ$ denotes functional composition, i.e. $(f \circ g)(p) = g(f(p))$.

Associated with any P_j in the *isa* hierarchy, we define P^j_{HP} to be the subset of P_j containing the clauses which are not erased by the subsetting performed to construct $HP_\lhd$. P^j_{HP} can be defined formally as follows:

$$P^j_{HP} \;=\; \{c \in P_j \mid \psi^j_{HP}(c) \in HP_\lhd\}$$

For any $p \in \pi(P^j_{HP})$, we call $\psi^j_{HP}(p)$ the predicate in $\pi(HP_\lhd)$ which *corresponds* to p. Pairs of corresponding predicates enjoy the following property: given $p \in P^j_{HP}$ each clause of the definition of $\psi^j_{HP}(p)$ in $HP_\lhd$ is a renaming of a clause selected for reducing the corresponding occurrence of p in HP. This property, proved in the following lemma, is crucial to show the equivalence between *isa* and $\lhd$-hierarchies.

Let G be an atomic goal and let $p = Pred(G)$. Call $select(p, P_j, HP)$ the set of clauses which can be selected in HP for reducing G starting from the component P_j, and $def(q, HP_\lhd)$ the set of clauses which define predicate q in $HP_\lhd$.

Lemma 2.8 *Let $HP = P_n$ isa P_{n-1} isa $\cdots$ isa P_1 and let $HP_\lhd$ be the corresponding differential program. Let $p \in \pi(P^j_{HP})$. Then, for all $k \leq n$,*

$$c \in P_k \;\&\; c \in select(p, P_j, HP) \iff c \in P^k_{HP} \;\&\; \psi^k_{HP}(c) \in def(\psi^j_{HP}(p), HP_\lhd)$$

Proof. *See appendix.* ■

The equivalence between *isa*-proofs for an *isa*-hierarchy and refutations for the corresponding $\lhd$-composite program can be now established in terms of the following result.

Lemma 2.9 *Let HP be the hierarchy* P_n isa P_{n-1} isa $\cdots$ isa P_1 *and* $HP_\lhd = P_n \lhd \ldots \lhd P_1$ *the corresponding differential program. Moreover, let G be an atomic goal such that* $Pred(G) \in \pi(P^j_{HP})$. *Then*

$$P_j, HP \vdash_\theta G \iff \psi^j_{HP}(G) \stackrel{\gamma}{\longmapsto}_{HP_\lhd} \Box$$

where $\vartheta_{|G} = \gamma$.

Proof. *We prove that*

$$P_j, HP \vdash_\theta G \iff \psi^j_{HP}(G) \stackrel{\theta}{\rightsquigarrow}_{HP_\lhd, LD} \Box$$

where LD is the left-to-right depth-first selection rule. Then the result follows by the independence from the selection rule for computed answers.

The proof follows by induction on the height of the isa*-proof on one side and the length of the LD-derivation on the other side. Assume that there exists a proof of height m for* $P_j, HP \vdash_\theta G$. *By lemma 2.8, a clause c is selected from* P_k *in the first step of the isa-proof if and only if* $c \in P^k_{HP}$ *and* $\psi^k_{HP}(c)$ *is selected by the first step of the LD refutation in* $HP_\lhd$. *If* $c = G' \text{:-} B_1, \ldots, B_n$ *and* $\theta_1 = mgu(G, G')$, *then the next step consists of proving* $P_k, HP \vdash_{\theta_2} (B_1, \ldots, B_n)\theta_1$, *with* $\theta = \theta_1\theta_2$, *in the isa-hierarchy and of solving the query* $(\psi^k_{HP}(B_1), \ldots, \psi^k_{HP}(B_n))\theta_1$ *in the differential program. In the isa-hierarchy, the proof will be split into n sub-proofs of height less or equal to* $m-1$. *Correspondingly, the derivation in* $HP_\lhd$ *can be split into n corresponding sub-derivations since, for all i,* $Pred(B_i) \in \pi(P^k_{HP})$ *and, by the inductive hypothesis,*

$$P_k, HP \vdash_{\theta_2} B_1\theta_1 \iff \psi^k_{HP}(B_1\theta_1) \stackrel{\theta_2}{\rightsquigarrow}_{HP_\lhd, LD} \Box$$

Now the claim follows by repeating the same argument n-times for each of the B_i*'s.* ■

Theorem 2.10 *Let HP be an isa-hierarchy and* $HP_\lhd$ *be the corresponding* $\langle \Sigma, \Delta, \Theta \rangle$*-differential program. Then for any goal G such that* $Pred(G) \subseteq (\Sigma \cup \Delta \cup \Theta)$*:*

$$HP \vdash_\vartheta G \iff G \stackrel{\gamma}{\longmapsto}_{HP_\lhd} \Box$$

where $\vartheta_{|G} = \gamma$.

Proof. *If G is atomic then, by definition of* $\vdash$, $HP \vdash_\theta G \iff P_k, HP \vdash_\theta G$ *where* $Pred(G) \in \kappa(P_k)$ *and k is determined as follows: if* $Pred(G) \in \Delta \cup \Sigma$ *then* P_k *is the top-most component of HP which defines p, while if* $Pred(G) \in \Theta$ *then* P_k *is any component of HP which defines p.*

We can apply lemma 2.9 provided that $Pred(G) \in \pi(P^k_{HP})$. *To see this observe that:*

(i) if $Pred(G) \in \Theta \cup \Sigma$ *then* $Pred(G) \in \pi(P^k_{HP})$ *because no definition for any* Θ *or* Σ *predicate is removed in the syntactic composition.*

(ii) if $Pred(G) \in \Delta$ *then* $\psi^k_{HP}(p) = p$ *for* P_k *is the top-most component of HP which defines* p.

Now, to complete the proof we have only to show that $\psi^k_{HP}(Pred(G)) = Pred(G)$. *This is obvious for* Θ *and* Δ *predicates. When* $Pred(G)$ *is a* Σ *predicate it follows because* P_k *is the top-most component of HP which defines* p.

In the case of conjunctive goals the proof follows immediately by induction. ■

The proof highlights one important property of the renamings used in the $\lhd$-composition, namely that the renamings preserve the predicates exported by the corresponding *isa*-composition. Hence, the $\langle \Sigma, \Delta, \Theta \rangle$-differential program $HP_\lhd$ and the corresponding *isa*-hierarchy *HP* prove exactly the same goals G provided that $Pred(G) \subseteq (\Sigma \cup \Delta \cup \Theta)$. Note that this condition is equivalent to assume that $Pred(G)$ does not contain internal predicates. In fact if $p \in Pred(G)$ is not internal and $p \notin (\Sigma \cup \Delta \cup \Theta)$, then G fails both in the hierarchy *HP* and in the corresponding program $HP_\lhd$.

3 A $\lhd$-compositional Semantics for Differential Programs

Having established the equivalence between *isa* hierarchies and $\lhd$-composite programs, we move on to study a $\lhd$-compositional semantics for the class of differential programs. The approach we follow is inspired by the work on the semantics of *open* logic programs developed in [3, 4] and [5]. Similarly to differential programs, open logic programs are understood as program components rather than stand-alone programs: their composition is performed taking the union of the components' clauses. The work on the semantics of open logic programs was motivated by the fact that the standard minimal-model semantics of logic programming is not *union*-compositional. To see this, consider the following classical example. Let $M(P)$ denote the least Herbrand model of P.

Example 3.1 *Let* $P_1 = \{r(a)\}$ *and* $P_2 = \{p(X) \text{:-} r(X),\ r(b)\}$ *be two programs. The semantics of the union of* P_1 *and* P_2 *is* $M(P_1 \cup P_2) = \{p(a), p(b), r(a), r(b)\}$. *It is immediate to see that* $M(P_1 \cup P_2)$ *cannot be obtained from* $M(P_1)$ *and* $M(P_2)$, *since* $M(P_1) = \{r(a)\}$ *and* $M(P_2) = \{p(b), r(b)\}$.

Needless to say, the minimal-model semantics is also non-compositional with respect to any other more complex operator, like our inheritance mechanisms, defined in terms of union. The same argument applies also to other semantics, such as those reported in [12], which are defined on sets of (non ground) atoms. The problem is that considering a program as part of a collection of programs, makes its meaning dependent on the context that program is part of. Given the interpretation of the context, the semantics of the program is a function of that interpretation. Functional semantics for logic program have been largely investigated in the literature: the semantics based on the T_P operator of [21] and on the closure operator $(T_P + id)^\omega$ of [18] are examples of such definitions. The semantics for *open* logic programs of [3, 4] and [5] is in some respects

similar to the definition based on the semantics $(T_P + id)^\omega$ of [18], but it refines it in that: *(i)* it captures the operational behaviour of programs more precisely (it is proved *cas*-correct) and (*ii*) it provides a syntactic representation of the semantics $(T_P + id)^\omega$, a kind of normal form representation for that functional semantics.

The idea, which motivated also the definition of the semantics proposed in [14], is to use sets of clauses as the semantic objects used to interpret a program. Roughly, an Ω-open program P is a program where the predicates contained in Ω are considered being partially defined in P. The *open* semantics $\mathcal{O}_{\Omega(P)}$ of an Ω-open program P, is given by the set of *resultants* [20] obtained in P starting from the most general form of the goals for all the predicate symbols in $\pi(P)$ and ending in resolvents containing only predicates in Ω. Formally,

$$\mathcal{O}_\Omega(P) = \{p(\tilde{X})\vartheta \text{:-} \tilde{B} \mid \ \exists\ R \ s.t.\ p(\tilde{X}) \overset{\vartheta}{\leadsto}_{P,R} \tilde{B} \text{ and } Pred(\tilde{B}) \subseteq \Omega\}$$

The intuition behind this definition is that the meaning of a program is the set of all the *partial* answers which can be obtained by derivations ending in "open" resolvents. A partial answer is represented semantically by including the resultant yielding that answer in the program's denotation. Having clauses in the denotation makes the program's meaning dependent on the context. In fact, under this definition, it is shown in [4] that the semantics $\mathcal{O}_\Omega(P)$ is *union*-compositional.

Theorem 3.2 *(Theorem 2.13 in [4]) Let $\Omega, \Omega_1, \Omega_2$ be sets of predicates symbols such that $\Omega \subseteq \Omega_1 \cup \Omega_2$. Let P_1 be an Ω_1-open program, P_2 be an Ω_2-open program such that $Pred(P_1 \cap P_2) \subseteq \Omega_1 \cap \Omega_2$, and let $\mathcal{O}_\Omega(P)$ be defined as above. Then*

$$\mathcal{O}_\Omega(P_1 \cup P_2) \ = \ \mathcal{O}_\Omega(\mathcal{O}_{\Omega_1}(P_1) \cup \mathcal{O}_{\Omega_2}(P_2))$$

Although adequate to model program composition by union, this semantic characterization is not adequate to capture the type of program composition we are proposing in this paper. To see this, consider the following example where we consider the Ω's as the set of all predicate symbols and hence omit them.

Example 3.3 *Let $\langle \Sigma_1, \Delta_1, \Theta_1 \rangle$-$P_1$ and $\langle \Sigma_2, \Delta_2, \Theta_2 \rangle$-$P_2$ be the following differential programs*

$$P_2 = \{r(a).\} \qquad P_1 = \{ \quad p(X) : -r(X). \\ \qquad\qquad\qquad\quad r(b). \qquad\qquad \}$$

where $\Delta_2 = \{r\}$, $\Delta_1 = \{r, p\}$ and $\Sigma_i = \Theta_i = \emptyset$ for $i = 1, 2$. The composition $P_2 \lhd P_1$ corresponds to the program $\{r(a),\ p(X)\text{:-}r(X)\}$ where the clause $r(b) \in P_1$ has been overridden by the clause $r(a) \in P_2$. Now, if we apply the previous definition, we obtain

$$\mathcal{O}_\Omega(P_1 \cup P_2) = \{r(b),\ p(b),\ r(a),\ p(a),\ p(X)\text{:-}r(X)\}.$$

Note that $\mathcal{O}_\Omega(P_1 \cup P_2)$ is a superset of what we expect as the semantics of $P_2 \lhd P_1$. The problem is that to obtain the semantics of $P_2 \lhd P_1$, from $\mathcal{O}_\Omega(P_1 \cup P_2)$ we should

delete from $\mathcal{O}_\Omega(P_1 \cup P_2)$, not only $r(b)$, as we expect as a consequence of the overriding semantics of $\lhd$, but also $p(b)$ which is derived from $r(b)$. Thus, when defining the semantics of P_1, we need a mechanism for recording that $p(b)$ has been obtained by using the definition of the Δ-predicate r, local to P_1, which could be overridden by the context. This is achieved by introducing the following notion of *context sensitive clause* as element of the semantic domain.

Definition 3.4 *A* context sensitive clause (cs-clause) *is an object of the form*

$$A \text{:-} \{q_1, \ldots, q_n\} \Box B_1, \ldots, B_k \tag{1}$$

where $\{q_1, \ldots, q_n\}$ *is the context of the cs-clause,* $q_1, \ldots, q_n$ *are predicate symbols and* $A, B_1, \ldots, B_k$ *are atoms.*

The intuitive meaning of (1) is that the logical implication $A \leftarrow B_1, \ldots, B_k$ is true in any context which does not override the definitions of $q_1, \ldots, q_n$. A standard clause can be seen as a cs-clause with an empty context. To simplify the notation, we will henceforth assume that empty contexts are not written explicitly. Accordingly, we will consider a clause as a special case of a cs-clause.

A *context sensitive interpretation* is defined in terms of equivalence classes of cs-clauses as follows. Say that two cs-clauses $c_1 = H_1$:- $s_1 \Box \tilde{B}_1$ and $c_2 = H_2$:- $s_2 \Box \tilde{B}_2$ are equivalent ($c_1 \approx c_2$) iff, $s_1 = s_2$ and, considering the $\tilde{B}_i$'s as multisets, c_1 and c_2 are equal up to variable renaming.

Definition 3.5 (cs-interpretation) *Let* $\mathcal{C}_\Delta$ *denote the set of all the* $\approx$*-equivalence classes of the cs-clauses* $A \text{:-} s \Box \tilde{B}$ *such that* $s \subseteq \Delta$. *A cs-interpretation for a* $\langle \Sigma, \Delta, \Theta \rangle$*-program* P *is any* $I \subseteq \mathcal{C}_\Delta$.

Remarks. In the following we denote the $\approx$-equivalence class of a cs-clause c by c itself. Abusing the notation, we will also identify syntactic operators on *cs*-clauses with (semantic) operators on $\mathcal{C}_\Delta$. The representatives of each equivalence class contained in $\mathcal{C}_\Delta$ will be assumed to be renamed apart from the elements in the class. This is consistent with the definition of the semantic operators which are given independently of choice of the representative of the equivalence class. The previous definitions for programs (such as $\Omega(P)$, $\iota(P)$ etc.) are implicitly extended to apply to cs-interpretations and the notion of differential programs is naturally extended to sets of cs-clauses. The context will be ignored when computing the answers substitutions. Finally, to simplify the notation, we will omit the prefix $\langle \Sigma, \Delta, \Theta \rangle$ for cs-interpretations when no ambiguity arises.

The semantics of a differential program is defined by a fixed point construction based on the immediate-consequence operator T_P^{cs} for *cs*-interpretations. T_P^{cs} is defined in terms of an unfolding rule. Recall that for a differential program P, the open predicates are the predicates symbols in the set $\Omega(P) = (\Sigma_P \setminus \kappa(P)) \cup \Delta_P \cup \Theta_P$. Given

any set of predicate symbols Ψ, we denote by Id_Ψ the set of cs-clauses (with empty context) $\{p(\tilde{X}) \text{:-} \ p(\tilde{X}) |\ p \in \Psi$ and $\tilde{X}$ distinct variables$\}$.

Definition 3.6 (T_P^{cs}) *Let P be a $\langle \Sigma, \Delta, \Theta \rangle$-program and let I be a cs-interpretation for P. Then we define*

$$T_P^{cs}(I) = unf_{P,\Omega(P),\Delta}(I \cup Id_{\Omega(P)}).$$

where, given two sets of predicate names Ψ and Δ, the cs-unfolding $unf_{P,\Psi,\Delta}$ is defined by

$$\begin{aligned} unf_{P,\Psi,\Delta}(I) = \{ \quad & A\theta \text{:-} s \cup C \cup C_1 \ldots \cup C_k \ \Box \ (\tilde{L}_1, \ldots, \tilde{L}_k)\theta \ \mid \\ & \exists\ A \text{:-} s \ \Box \ B_1, \ldots, B_k \in P, \\ & \exists\ cl_i = B_i' \text{:-} C_i \Box \tilde{L}_i \in I,\ i = 1, \ldots, k, \\ & \theta = mgu((B_1, \ldots, B_k), (B_1', \ldots, B_k')), \\ & C = \{Pred(B_i) \ \mid \ Pred(B_i) \in \Delta \ \textit{and} \ \ cl_i \notin Id_\Psi\} \ \} \end{aligned}$$

The intuition behind the definition of T_P^{cs} is the following: whenever we unfold an atom B_i in a cs-clause we add $Pred(B_i)$ to the context of that cs-clause if and only if $Pred(B_i)$ is a Δ-predicate and the cs-clause used to unfold B_i is not in $Id_{\Omega(P)}$. The Δ-predicate $Pred(B_i)$ is recorded in the context of the cs-clause produced by the unfolding step so that the context of that cs-clause can be used to model semantically the overriding semantics of the $\lhd$ composition. This argument does not apply to the case when a Δ-predicate is unfolded using a clause in $Id_{\Omega(P)}$ because in the definition of $T_P^{cs}(I)$ the clauses in $Id_{\Omega(P)}$ are added to I only to "delay" the evaluation of open predicates.

Proposition 3.7 (Continuity) *T_P^{cs} is continuous on the complete lattice $(\mathcal{C}_\Delta, \subseteq)$.*

The continuity of T_P^{cs} follows directly from the continuity of the corresponding operator introduced in [3, 4]. Note, to this regard, that when all the clauses in the cs-interpretations have empty contexts, T_P^{cs} coincides with the operator used in the definition of the open semantics of [3, 4]. Now, since T_P^{cs} is continuous on the complete lattice $(\mathcal{C}_\Delta, \subseteq)$, the least fixed point of T_P^{cs} can be computed as $T_P^{cs} \uparrow \omega$.

The semantics of any differential program P will be defined as an abstraction of $T_P^{cs} \uparrow \omega$. There are two levels at which $T_P^{cs} \uparrow \omega$ can (and in fact should) be abstracted upon. Firstly consider the internal predicates of P. Since the predicates $\iota(P)$ are considered internal to the program, any cs-clause defining them should not be part of the semantics of P. Excluding these definitions corresponds to ensure that internal predicates are not exported, sematically speaking.

Secondly, assume that $T_P^{cs} \uparrow \omega$ contains two cs-clauses c and c' which differ only in that set of constraints (the context) s_c of c is a subset of the context $s_{c'}$ of c'. Let, for instance, $s_{c'}$ be the empty set and $s_c = \{q\}$. Now c and c' bear the exact same meaning as far as the semantics of P is concerned. The difference is that c will get erased from the semantics of any composition $Q \lhd P$ where Q which redefines q, whereas c' will be

part of it. Hence, when computing the sematics of P we can safely drop c as long as we retain c'. This intuitive picture giustifies the following definition of the abstraction α over cs-interpretations.

Definition 3.8 *Let I be a set of (equivalence classes of) cs-clauses. The abstraction $\alpha(I)$ is defined as follows:*

$$\alpha(I) = \{H :\!- s \Box \tilde{B} \in I \mid \nexists H' :\!- s' \Box \tilde{B}' \in I \text{ such that } s' \subset s,\ H :\!- \tilde{B} \approx H' :\!- \tilde{B}'\}.$$

For any two cs-interpretations I and J, we then then define $I =_\alpha J$ iff $\alpha(I) = \alpha(J)$. Finally, given a set Ψ of predicate symbols we denote with $\mathcal{D}(\Psi)$ the set of cs-clauses whose head's predicate symbol belongs to Ψ. Formally: $\mathcal{D}(\Psi) = \{H :\!- s \Box \tilde{B} \mid Pred(H) \in \Psi\}$.

Definition 3.9 (Fixpoint semantics) *Let P be a $\langle \Sigma, \Delta, \Theta \rangle$-program. The fixpoint semantics $[\![P]\!]$ of P is defined as follows:*

$$[\![P]\!] = \langle \Sigma_{[\![P]\!]}, \Delta_{[\![P]\!]}, \Theta_{[\![P]\!]} \rangle\text{-}\alpha(T_P^{cs} \uparrow \omega) \setminus \mathcal{D}(\iota(P)).$$

The semantics $[\![P]\!]$ of P is again considered a differential program. As such, according to the definition of differential programs, the annotation of $[\![P]\!]$ is obtained by intersecting the original annotation $\langle \Sigma, \Delta, \Theta \rangle$ of P with the set $\pi([\![P]\!])$ of predicate symbols of $[\![P]\!]$. Notice finally that from definition 3.9, it follows that if $H :\!- s \Box \tilde{B} \in [\![P]\!]$ then $Pred(\tilde{B}) \subseteq \Omega(P)$ and this makes $[\![P]\!]$ a function of the *open* predicates of P.

Example 3.10 *The semantics of the programs of example 2.5 are the following*

$$[\![P]\!] = \begin{cases} s(a). & q(a). \\ r(c). & t(b). \end{cases}$$

$$[\![Q]\!] = \begin{cases} s(x) :\!-\ r(x). & h(x) :\!-\ t(x). \\ s(a) :\!- \{r\} \Box\ q(x). & s(a). \\ r(a) :\!-\ q(x). & \end{cases}$$

$$[\![R]\!] = \begin{cases} q(a). & r(b). \\ h(a). & s(x) :\!-\ t(x). \end{cases}$$

The intuitive meaning of the cs-clause $s(a) :\!- \{r\} \Box q(x)$, is that the implication $s(a) \leftarrow q(x)$ holds true as long as the context in which Q occurs does not override the definition for r local to Q. If Q occurs in the context $P \lhd Q$ where P's definition of r is the unit clause $r(c).$, then the new definition of r in P overrides clause $r(a) :\!- q(x)$ thus invalidating the constrained implication $s(a) :\!- \{r\} \Box q(x)$.

3.1 Correctness

We now prove the adequacy of our semantics with respect to the operational semantics defined in terms of *isa*-proofs. This is accomplished in two steps: we first show, in theorem 3.12, that $[\![\cdot]\!]$ is (cas)-correct, i.e. it models correctly the answer-substitution semantics of our programs; then, in theorem 3.13, we use the equivalence between *isa*-hierarchies and $\lhd$-composite programs to extend the correctness result to *isa*-hierarchies

The first result follows by considering the relation between the semantics $[\![\cdot]\!]$ and the s-semantics defined in [12]. The s-semantics of a program P is defined as the least fixed point of a continuous operator T^s_P defined on sets of (equivalence classes of) atoms. Similarly to T^{cs}_P, T^s_P is defined in terms of a corresponding unfolding operator, unf^s_P, as follows: $T^s_P(I) = unf^s_P(I)$. The difference is that, while T^{cs}_P is a function of sets of (non ground) clauses, the domain of T^s_P (and unf^s_P) is the power-set of an extended Herbrand base whose elements are non ground atoms. As a matter of fact, it is easy to show that $unf_{P,\Psi,\Delta}$ is a generalization of unf^s_P. In fact, the definition of unf^s_P can be obtained from that of $unf_{P,\Psi,\Delta}$ by choosing $\Psi = \Delta = \emptyset$ and by restricting the domain to sets of atoms. More formally: for any $I \subseteq \mathcal{C}_\Delta$, let $I_{|s}$ denote the set $I_{|s} = \{A \mid A\text{:-}s \ \Box \in I\}$. We can show that

$$unf^s_P(I_{|s}) \quad = \quad (unf_{P,\Psi,\Delta}(I))_{|s}$$

The strong completeness theorem for s-semantics states that any computed answer for the goal G in P can be obtained by "evaluating" G in the s-semantics of P.

Theorem 3.11 *[12] Let P program and let $G = A_1, \ldots, A_k$ be a goal . Then*

$$\begin{array}{rcl} G \stackrel{\vartheta}{\longmapsto}_P \Box & \Longleftrightarrow & \exists\ H_i \in T^s_P \uparrow \omega,\ i = 1, \ldots, k, \\ & & \exists \gamma = mgu((A_1, \ldots, A_k)(H_1, \ldots, H_k)) \\ & & \gamma_{|G} = \vartheta \end{array}$$

The corresponding result for our semantics $[\![\cdot]\!]$ for differential programs can now be stated as follows:

Theorem 3.12 (correctness) *Let P be a $\langle \Sigma, \Delta, \Theta \rangle$-program and let $G = A_1, \ldots, A_k$ be a goal with $Pred(G) \subseteq (\Sigma \cup \Delta \cup \Theta)$. Then*

$$\begin{array}{rcl} G \stackrel{\vartheta}{\longmapsto}_P \Box & \Longleftrightarrow & \exists\ H_i\text{:-}s_i \ \Box \in [\![P]\!],\ i = 1, \ldots, k, \\ & & \exists \gamma = mgu((A_1, \ldots, A_k)(H_1, \ldots, H_k)) \\ & & \gamma_{|G} = \vartheta \end{array}$$

Proof. *From the relation between unf^s_P and $unf_{P,\Psi,\Delta}$, it follows that, given any cs-interpretation I, $T^s_P(I_{|s}) = (T^{cs}_P(I))_{|s}$. Then, by a straightforward inductive argument, we have that $H\text{:-}s \ \Box \in T^{cs}_P \uparrow n$ iff $H \in T^s_P \uparrow n$ for any n. Therefore, by definition of $\uparrow$ and by definition of $[\![\cdot]\!]$, if $Pred(H) \in \Sigma \cup \Delta \cup \Theta$, $H\text{:-}s \ \Box \in [\![P]\!]$ iff $H \in T^s_P \uparrow \omega$. In other words, $[\![P]\!]$ contains an isomorphic copy of the s-semantics for the open predicates of P. Since by hypothesis $Pred(G) \subseteq (\Sigma \cup \Delta \cup \Theta)$ the thesis holds by theorem 3.11.* ■

To prove the adequacy of our semantics with respect to the operational semantics defined in terms of *isa*-proofs, we can now use the correspondence between computations in *isa* hierarchies and computations in $\lhd$ programs, as stated by theorem 2.10.

Theorem 3.13 *Let HP be an isa-hierarchy, $HP_\lhd$ be the corresponding $\langle\Sigma,\Delta,\Theta\rangle$-program and $G = A_1,\ldots,A_k$ be a goal with $Pred(G) \subseteq (\Sigma \cup \Delta \cup \Theta)$. Then:*

$$HP \vdash_\vartheta G \iff \begin{array}{l} \exists\ H_i\text{:-}s_i\ \square \in [\![HP_\lhd]\!],\ i = 1,\ldots,k, \\ \exists \gamma = mgu((A_1,\ldots,A_k)(H_1,\ldots,H_k)), \end{array}$$

where $\gamma_{|G} = \vartheta_{|G}$.

Proof. *Immediate from theorems 2.10 and 3.12.* ■

As a corollary, we can prove that semantic equality between $\lhd$-hierarchies implies sameness of answer substitutions on *isa*-hierarchies. Call $\approx^{cas}$ the *observational* equivalence, based on answer substitutions, for *isa*-hierarchies:

$$HP \approx^{cas} HP' \iff HP \vdash_\vartheta G \text{ iff } HP' \vdash_{\vartheta'} G$$

where $\vartheta_{|G} = \vartheta'_{|G}$.

Corollary 3.14 *Let HP and HP′ be isa-hierarchies and let $HP_\lhd$ and $HP'_\lhd$ be the corresponding $\langle\Sigma,\Delta,\Theta\rangle$-programs. Then:*

$$[\![HP_\lhd]\!] = [\![HP'_\lhd]\!] \ \Rightarrow\ HP \approx^{cas} HP'$$

Proof. *Immediate from theorem 3.13.* ■

3.2 Compositionality

We now show the $\lhd$-compositionality of the semantics $[\![\cdot]\!]$. We first introduce a semantic operation $\prec$ on cs-interpretations which corresponds to the syntactic $\lhd$-composition of differential programs. As for $\lhd$, the operator $\prec$ is considered to be right-associative.

Definition 3.15 *Let $\langle\Sigma_P,\Delta_P,\Theta_P\rangle$-$P$ and $\langle\Sigma_Q,\Delta_Q,\Theta_Q\rangle$-$Q$ be compatible differential programs and let $[\![P]\!]$ and $[\![Q]\!]$ the respective semantics. We define the semantic composition $[\![P]\!] \prec [\![Q]\!]$ as $[[\![P]\!]^l \cup [\![Q]\!]^r]$ where*

$$[\![P]\!]^l = \{A\text{:-}s\ \square\ \tilde{B} \in [\![P]\!] \mid\ Pred(\tilde{B}) \cap \Sigma_P \subseteq \kappa([\![Q]\!]^r)\}$$

$$[\![Q]\!]^r = \{A\text{:-}s\ \square\ \tilde{B} \in [\![Q]\!] \mid\ \begin{array}{l} s \cap \kappa(P) = \emptyset,\ Pred(\tilde{B}) \cap \Sigma_Q = \emptyset \\ Pred(A) \notin \kappa(P) \cap (\Sigma_Q \cup \Delta_Q) \end{array} \}$$

The definition of $[\![P]\!] \prec [\![Q]\!]$ is given along the same guidelines of the corresponding definition for the syntactic $\lhd$-composition. The intuition is the following. First recall that all the cs-clauses in $[\![P]\!]$ and $[\![Q]\!]$ are the result of the unfolding process on P and

Q. Now, take a cs-clause c in $[\![P]\!]$ whose body contains an atom b such that $Pred(b)$ belongs to Σ_P. From the definition of the fixpoint semantics, it follows that $Pred(b)$ is not defined by P. Then, if $[\![Q]\!]^r$ does not contain any definition for $Pred(b)$, c can be deleted from $[\![P]\!] \prec [\![Q]\!]$. As for the syntactic composition, the deletion is safe in this case being $\lhd$, and hence $\prec$, assumed to be right-associative.

The same argument motivates the corresponding condition on $[\![Q]\!]^r$. The remaining condition on $[\![Q]\!]^r$ provide the semantic counterpart of the overriding that occurs at the syntactic level between P and Q. Recall the two programs of example 3.3. We said that to compute the semantics of $P \lhd Q$ in a compositional way, we should have deleted from the semantics of Q, not only the definition of the Δ-predicate r, but also everything derived in Q using r's definition. The two conditions given above on $[\![Q]\!]^r$ model precisely this mechanism. Note also that $[\![P]\!]^l \prec [\![Q]\!]^r$ can be considered a $\langle \Sigma, \Delta, \Theta \rangle$-program, where the annotation $\langle \Sigma, \Delta, \Theta \rangle$ is obtained according to the usual restrictions for differential programs. Namely, $\Sigma = (\Sigma_P \cup \Sigma_Q) \cap \pi([\![P]\!] \prec [\![Q]\!])$ and similarly for Δ and Θ. A final note concerns the fact that the definition of $[\![P]\!] \prec [\![Q]\!]$ depends also on a piece of syntactic information ($\kappa(P)$) and hence, strictly speaking, $\prec$ is not a purely semantic operator. However, it is easy to see that this could have been avoided by embedding this information into the semantics $[\![\cdot]\!]$.

The proof of the $\lhd$-compositionality of $[\![\cdot]\!]$ relies on tight relation existing between the syntactic $\lhd$-composition of programs and the semantic $\prec$-composition of cs-interpretations. Due to the correspondence between the conditions which define the sets $[\![P]\!]^l$, $[\![Q]\!]^r$ and $\xi(P)$, $\phi(Q^\star)$ we can show that taking the semantics of $[\![P]\!]^l \cup [\![Q]\!]^r$ is equivalent to taking the semantics of $\xi(P) \cup \phi(Q^\star)$. Then the result follows by observing that $[\![P]\!] \prec [\![Q]\!] = [\![[\![P]\!]^l \cup [\![Q]\!]^r]\!]$ and correspondingly, $[\![P \lhd Q]\!] = [\![\xi(P) \cup \phi(Q^\star)]\!]$.

We first need the two following lemmas whose proofs are reported in the appendix.

Lemma 3.16 *Let $\xi(P)$ and $\phi(Q^\star)$ be defined according to definition 2.4. Then*

$$[\![\xi(P) \cup \phi(Q^\star)]\!] \quad = \quad [\![[\![\xi(P)]\!] \cup [\![\phi(Q^\star)]\!]]\!]$$

Lemma 3.17 *Let $\langle \Sigma_P, \Delta_P, \Theta_P \rangle$-$P$,$\langle \Sigma_Q, \Delta_Q, \Theta_Q \rangle$-$Q$ be differential programs, $P \lhd Q$ be the differential program $\langle \Sigma, \Delta, \Theta \rangle$-$(\xi(P) \cup \phi(Q^\star))$ where $Q^\star$, ξ, ϕ is defined according to definition 2.4. Moreover let $[\![Q]\!]^r$ be defined according to definition 3.15. Then*

1. $[\![\xi(P)]\!] = [\![P]\!] \setminus R$ *where* $R = \{H \,\text{:-}s\, \square\, \tilde{B} \in \mathcal{C}_\Delta \mid Pred(\tilde{B}) \cap \Sigma_P \not\subseteq \kappa(Q)\}$
2. $[\![\phi(Q^\star)]\!] = [\![Q]\!]^r$

Theorem 3.18 (compositionality) *Let $\langle \Sigma_P, \Delta_P, \Theta_P \rangle$-$P$ and $\langle \Sigma_Q, \Delta_Q, \Theta_Q \rangle$-$Q$ be differential programs. Then*

$$[\![P \lhd Q]\!] \quad = \quad [\![P]\!] \prec [\![Q]\!]$$

Proof. *We first show that*

$$[\![[\![\xi(P)]\!] \cup [\![\phi(Q^\star)]\!]]\!] \quad = \quad [\![[\![P]\!]^l \cup [\![\phi(Q^\star)]\!]]\!] \tag{1}$$

To prove this, we proceed as follows. From lemma 3.17.1 we have that

$$[\![\xi(P)]\!] = [\![P]\!] \setminus R \text{ where } R = \{H \text{:-} s \ \Box\tilde{B} \mid Pred(\tilde{B}) \cap \Sigma_P \not\subseteq \kappa(Q)\}.$$

Now, since by definition 3.15, $[\![P]\!]^l = \{A \text{:-} s \ \Box \ \tilde{B} \in [\![P]\!] \mid Pred(\tilde{B}) \cap \Sigma_P \subseteq \kappa([\![Q]\!]^r)\}$, *we have that:*

$$[\![P]\!]^l \quad = \quad [\![\xi(P)]\!] \setminus \{H \text{:-} s \ \Box \ \tilde{B} \mid \exists p \in Pred(\tilde{B}) \cap \Sigma_P \ s.\ t.\ p \in \kappa(Q) \setminus \kappa([\![Q]\!]^r)\} \quad (2)$$

Let assume now that $c = H \text{:-} s \ \Box \ \tilde{B} \in [\![\xi(P)]\!] \setminus [\![P]\!]^l$. *Then there exists* $p \in Pred(\tilde{B}) \cap \Sigma_P$ *with* $p \in \kappa(Q) \setminus \kappa([\![Q]\!]^r)$. *It is easy to verify that, since* $\phi(p) = p \in \kappa(\phi(Q^\star))$, $p \notin \Omega(\xi(P) \cup \phi(Q^\star))$. *Then by definition of* $[\![\cdot]\!]$, $c \notin [\![[\![\xi(P)]\!] \cup [\![\phi(Q^\star)]\!]]\!]$ *and, since* $p \notin \kappa([\![\xi(P)]\!] \cup [\![\phi(Q^\star)]\!]) \cup \Omega(\xi(P) \cup \phi(Q^\star))$, *c cannot be used to derive new clauses in* $[\![[\![\xi(P)]\!] \cup [\![\phi(Q^\star)]\!]]\!]$. *Hence the claim follows from (2).*

Now we can reason as follows:

$$\begin{array}{rcl}
[\![P]\!] \prec [\![Q]\!] & \stackrel{def}{=} & [\![[\![P]\!]^l \cup [\![Q]\!]^r]\!] \\
\textit{by lemma 3.17.2} & = & [\![[\![P]\!]^l \cup [\![\phi(Q^\star)]\!]]\!] \\
\textit{by (1)} & = & [\![[\![\xi(P)]\!] \cup [\![\phi(Q^\star)]\!]]\!] \\
\textit{by lemma 3.16} & = & [\![\xi(P) \cup \phi(Q^\star)]\!] \\
\textit{by definition} & = & [\![P \lhd Q]\!]
\end{array}$$

■

The following example illustrates the compositional construction of the semantics.

Example 3.19 *Let's consider programs Q and R introduced in example 2.5 and their respective semantics (example 3.10). From definition 3.15,* $[\![Q]\!] \prec [\![R]\!] = [\![[\![Q]\!]^l \cup [\![R]\!]^r]\!]$ *where* $[\![Q]\!]^l \cup [\![R]\!]^r$ *is given by the cs-interpretation*

$$\left\{ \begin{array}{ll} s(x) \text{:-} r(x). & s(a). \\ s(a) \text{:-} \{r\}\Box q(x) & q(a). \\ r(a) \text{:-} q(x). & \end{array} \right.$$

Note that, according to definition 3.15, clause $h(a) \in [\![R]\!]$ *does not appear in* $[\![R]\!]^r$ *because the predicate* $h \in \Sigma_R$ *is defined in Q. Clause* $s(x) \text{:-} t(x) \in [\![R]\!]$ *does not appear in* $[\![R]\!]^r$ *because* $t \in \Sigma_R$ *is not defined in R. Also* $r(b) \in [\![R]\!]$ *is deleted since* $r \in \Delta_R$ *and r is defined in Q. Correspondly,* $[\![Q \lhd R]\!]$ *is the cs-interpretation:*

$$\left\{ \begin{array}{ll} s(x) \text{:-} r(x). & s(a). \\ r(a). & q(a). \end{array} \right.$$

Note that, since q is static and defined in $Q \lhd R$, *in the semantics of* $Q \lhd R$ *there are no cs-clause with q in the body. Moreover observe that the cs-clause* $s(a) : -\{r\}\Box$ *has been deleted due to the abstraction operator* $\alpha(\ )$. *It's easy to verify that the equality* $[\![Q \lhd R]\!] = [\![Q]\!] \prec [\![R]\!]$ *holds.*

Equivalence induced by $[\![\cdot]\!]$. We conclude this section studying the notion of observational equivalence induced by the *isa*-composition of programs and its relation with the equivalence induced by the semantics $[\![\cdot]\!]$.

We say that two differential programs P and Q are observationally equivalent with respect to the *isa* composition (and we write $P \approx^{cas}_{isa} Q$) if and only if P and Q can be interchanged in any *isa*-hierarchy without affecting the observational behaviour of that hierarchy. Since we are assuming that *isa* is right associative, the equivalence $\approx^{cas}_{isa}$ can be defined as follows:

Definition 3.20 *Let R and Q be two $\langle \Sigma, \Delta, \Theta \rangle$-differential programs. Then,*

$$R \approx^{cas}_{isa} Q \iff (P_n\ isa(\cdots isa\ R\ isa \cdots P_1)) \approx^{cas} (P_n\ isa(\cdots isa\ Q\ isa \cdots P_1))$$

for any choice of differential programs $P_1, \ldots, P_n$.

As for the semantics $[\![\cdot]\!]$, we have shown that it is *cas*-correct and $\lhd$-compositional. Now we can conclude that it is also *isa*-compositional, that is that for any two $\langle \Sigma, \Delta, \Theta \rangle$-differential R and Q, $[\![R]\!] = [\![Q]\!]$ implies that $R \approx^{cas}_{isa} Q$. Take R and Q such that $[\![R]\!] = [\![Q]\!]$ and let $P_1, \ldots, P_n$ be arbitrary differential programs. Then:

$$\begin{array}{rcl}
[\![R]\!] = [\![Q]\!] & \Longrightarrow & [\![P_n]\!] \prec \cdots [\![R]\!] \prec \cdots [\![P_1]\!] = [\![P_n]\!] \prec \cdots [\![Q]\!] \prec \cdots [\![P_1]\!] \\
\text{(by theorem 3.18)} & \Longleftrightarrow & [\![P_n \lhd \cdots R \lhd \cdots P_1]\!] = [\![P_n \lhd \cdots Q \lhd \cdots P_1]\!] \\
\text{(by theorem 3.14)} & \Longrightarrow & P_n\ isa\ \cdots R\ isa\ \cdots P_1 \approx^{cas} P_n\ isa\ \cdots Q\ isa\ \cdots P_1 \\
\text{(by definition)} & \Longleftrightarrow & R \approx^{cas}_{isa} Q
\end{array}$$

4 Applications

In this section we illustrate on a concrete example the application of the programming discipline we have discussed in the paper. The purpose of the following discussion is not to make a point in favour of inheritance as a programming methodology: the benefits of this approach to program development have long been recognized and fruitfully experienced in several applications.

Instead, what we wish to emphasize here is that not only the notions of specialization and refinement are amenable to be embedded into logic programming, but also that they can be exploited more naturally and effectively than in other programming paradigms. In particular, the use of extensible predicates introduces a type of specialization that has no counterpart in the traditional Object-Oriented frameworks. In fact, if refining a function, or a method in general, may only be achieved by (partially) *overriding* its definition, a natural and meaningful way to specialize a predicate is by *extending* the set of clauses which define it. The following example illustrates these issues more fully.

An Editor Project. We show how the project of `EMAX`, an Emacs-like editor, would be approached using the differential logic programming approach.

We make the assumption that the system supports the basic functionalities needed in the design of a display editor: buffer management, cursor moves, ... etc. Assuming Prolog as the underlying logic language at our disposal, we start by defining a generic interface module `EDITOR` providing the necessary machinery to make these primitives available as Prolog built-in predicates.

```
EDITOR
   open(File, Buff) :- ..... %% Associate a memory buffer with File
   save(Buff,File)  :- ..... %% Save the contents of buffer on File
   move(Buff, 'up') :- ..... %% Cursor moves
               ..................
```

`EMAX` is defined as specialization of `EDITOR` and it is conceived as a set of event-handlers for events coming from the associated editing window. Typing a character or clicking a mouse button on the editing window are typical examples of window events. The display manager collects the window events and serves them one at the time, by forwarding a corresponding query to `EMAX`. For the purpose of this example we will concentrate only on keystroke events and the associated handler `ks_event(Buff, L)`. `EMAX` distinguishes two classes of keystroke events depending on the list of characters `L` associated with each query `ks_event(Buff, L)`. The list `L` may either be initiated by a control character to request an editing function like search, cursor move, save ... etc, or consist of a single character to be echoed on the editing window. We will assume that some characters, typically parentheses, have a special treatment: besides echoing them, `EMAX` checks also whether they are balanced or not. As for the control requests we will consider only those initiated by the pattern `C-x`[1]. Finally `EMAX` defines a special handler for unexpected events such as system crashes. We will assume that unexpected events raise exceptions that are forwarded to `EMAX` as goals of the form `exception(Buffer, cause)`.

```
EMAX is_a EDITOR

%%%%%  Normal Events
   ks_event(Buff, ['C-x'|X]) :- !, cx_action(Buff, X).
   ks_event(Buff, [C]) :- echo(C, Buff),
                          match(Buff, C).
%%%%%  Handlers
   cx_action(Buff, 'C-c') :- get_Filename(Buff, File),
                             exit(Buff, File).
   cx_action(Buff, 'k') :- kill(Buff).
   exit(Buff, File) :- query_user('save changes ?', Ans),
                       Ans = 'yes',
                       save(Buff, File),         %% inherited
                       quit.
```

[1] `C-x` is the standard emacs abbreviation for the sequence of keystrokes "hold `CTRL` and type `x`"

```
   exit(Buff, File) :- quit.
%%%%% Brace checks
   match(Buff, '}') :- find_matching('{', Buff, Pos), !,
                       highlight(Pos, Buff).
   match(_, '}') :- !, warning('mismatched brace').
   match(_, C).
%%%%% Special Handlers
   exception(Buff, crash) :- save(Buff, *Buff*), %% inherited
                             quit.
```

The behaviour of the handler `cx_action` for `C-x` events should be obvious from the definition. `match/2` checks that a closed brace matches a corresponding open brace: if so it highlights the matching brace, otherwise it issues a warning message. The crash handler saves the current contents of `Buff` on the *auto-save* file `*Buff*` associated with `Buff`.

Let's now consider the signatures of the predicates defined in `EDITOR` and `EMAX`. `ks_event/2` is typically an extensible predicate: further specializations of `EMAX` might be instructed to provide special treatment for control sequences other than those supported by `EMAX`. Similar considerations apply to the balance checks defined by `match/2` and to `exception/2` which are thus assumed to be extensible.

A different way that `EMAX` may be specialized is by associating different responses to the `C-x` class of events. As a matter of fact, we may still want to use or extend the `cx_actions` defined by `EMAX` but, at the same time, we might very well want to modify part of their behaviour. With this idea in mind, we define `cx_action` as extensible and `exit/2` to be a dynamic (Δ) predicate. `open/2` and `move/2` are typical examples of inherited predicates for `EMAX` and, for the purpose of this example will be assumed to be static (Σ) predicates. As for `save/2`, we define it as static with the following idea: for any further specialization to be able to use a new saving routine upon exiting or upon a system crash we impose that it provides also new definitions for `exit/2` and `exception/2`. The following specialization of `EMAX` motivates this choice.

```
RCS-EMAX is_a EMAX

%%%% Redefine the exit routine.
   exit(Buff, File) :-  get_version_num(File,Vn),
                        save(Buff, File:Vn),
                        quit.
%%%% Save only changes since last version
   save(Buff, File:N) :- diff(Buff, File:N, DL),
                         N1 is N+1,
                         save_changes(DL, File:N1).
```

`RCS-EMAX` integrates the editing facilities supported by `EMAX` with revision control func-

tionalities supported by RCS[2]. Upon exiting from an editing session `RCS-EMAX` saves in a new *revision* of the file only the changes which have been made since the last time that file was edited. Hence, an exit request for `RCS-EMAX` is served, as expected, by the handler `cx_action/2` defined in `EMAX`; `cx_action/2`, in turn, activates the exit procedure and the associated save routing defined for `RCS-EMAX`.

On the contrary, `RCS-EMAX` delegates the treatment of exceptions to `EMAX`: an exception forwarded to `RCS-EMAX` activates the definition of `exception/2` in `EMAX` and this, in turn, a standard save on the auto-save file associated with the buffer being edited. Hence, there's is no attempt to save a (possibly inconsistent) new revision upon a system crash.

As another, independent, specialization consider the following module implementing a LaTeXmode for `EMAX`. `LaTeX-EMAX` extends the balance checks to characters like `$` as well as the class of control patterns associated with the keystroke events supported by `EMAX`

```
LaTeX-EMAX is_a EMAX

%%%% New class of events
   ks_event(Buff, ['C-c'|X]) :- cc_action(Buff, X).
%%%% Treatment of LaTeX environments
   cc_action(Buff, 'C-f') :- get_open_env(Buff, E),
                             put(Buff, nl),
                             put(Buff, '\end{E}').
%%%% New balance checks
   match(Buff, '$') :- find_prev('$', Buff, Pos),
                       highlight(Pos, Buff).
   match(_, '$') :- !, error('mismatched $').
```

Note that `LaTeX-EMAX` can be used either to specialize `EMAX` or as a futher specialization of `RCS-EMAX` with the obvious consequences on the treatment of multiple revisions.

5 Related Work

The work on modular extensions of logic programming was originally inspired by the proposal of R. O'Keefe in [28]. His idea was to give a formal account of one of the fundamental principles of the software engineering view of programming: programs should be developed incrementally by defining several units together with their interfaces and then by composing those units. This led him to propose a modular approach to programming based on the notion of program composition. He formalized this idea

[2]RCS is the Revision Control System developed by W. F. Tichy (see [31])

by interpreting logic programs as elements of an algebra and by modeling their composition in term of the operators of the algebra. The distinguishing property of this approach is that it extends logic programming with modular constructs without any need to extend the language of Horn clauses. In fact, module-composition is inherently a *meta-linguistic* mechanism. This idea gave way to the development of several proposals of modular systems based on the idea of program composition. The approach discussed by Bossi et al. in [3, 4] together with the compositional frameworks of Mancarella and Pedreschi ([21]), Gaifman and Shapiro ([14]), and of Brogi et al. [6] can in fact be seen as different formulations of this idea.

The novelty of the proposal presented in this paper is in the type of composition mechanisms we have considered as well as in the domain chosen for the semantic characterization. The *isa*-composition of differential programs provides a uniform semantics for the existing composition mechanisms and extends them with an explicit treatment of overriding inheritance which was missing in the aforementioned proposals. The use of *internal* predicates provides also a formal account of information hiding richer than those proposed by Gaifman and Shapiro in [14]. Finally, our approach represents the first attempt to capture, in a compositional fashion, a computational semantics of inheritance systems stated in terms of computed-answer-substitutions.

A different approach to the definition of a modular extension of logic programming was instead motivated by the idea of instrumenting logic programming with *linguistic* mechanisms for abstraction richer than those offered by Horn clauses. The idea was to provide a richer support for programming-in-the-small and then to tailor those mechanisms to attack the problems of programming-in-the-large. This approach originated with the work of D. Miller, in [22]. His idea was to allow implications to occur in the bodies of clauses and to use the deduction theorem to define a proof procedure for the extended language. Simply, he defined an implication goal $D \supset G$ to be provable in a program P if G is provable in the extended program $P \cup \{D\}$. The idea of modularity derives then by observing that, if D is a conjunction of clauses, we can interpret the goal $D \supset G$ as a scoping construct which requires that the clauses in D be loaded before evaluating G and then unloaded after G succeeds or fails. Implication goals as structuring tools were then used by a number of other authors in the attempt to capture more powerful scoping and modular constructs than those introduced by Miller (see for instance [27] and [15]).

Although different in their motivations, the two approaches are actually strictly related. In fact, the composition mechanisms are conceptually the same; the difference is that they act as meta-linguistic operators in the former and a linguistic operators on the latter (see [9] for a fuller discussion on this issue).

In [23] Monteiro and Porto proposed Contextual Logic Programming (CxLP) as a modular logic programming language based on a new type of implication goal, called *extension goal* and denoted by $D \gg G$. Operationally, $D \gg G$ is provable in the program P if (the goal) G can be proved in (the set of clauses) $D \cup A$, where A is a

finite set of atoms for predicates not defined in D and which can be proved in P. Thus the operator $\gg$ provides a *context extension*, i.e. a kind of lexical scoping which has essentially the same semantic connotation as *static* inheritance (while the implication goal $D \supset G$ is similar to *dynamic* inheritance). In a more recent paper ([26]) CxLP has been extended by introducing also a restricted form of dynamic scoping.

The major difference with our approach is that in the CxLP language the composition of different components occurs dynamically as the result of evaluating a query. In effect, CxLP's context extension, by providing a mechanism for dynamically specifying (and modifying) a unit's hierarchical links with its ancestors, captures a notion which is known as delegation [32]. Therefore, the compositional semantics of CxLP (and extensions thereof) introduced in [26] is based on a functional notion of denotation which associate to each unit u of the system a functional I_u whose domain is a set of functions on Herbrand intepretations.

In our case, the denotation of a unit is a set of cs-clauses obtained by a least fixpoint construction. An advantage of this characterization is that the standard abstract interpretation techniques used for logic programs (see [11] for a survey) can be applied to derive (compositional) methods for the analysis of differential programs. Moreover, differently from our case, the semantics [26] is intended to capture the model-theoretic meaning of a system an hence does not capture the notion of computed answer substitution.

In a related paper [24] Monteiro and Porto take a more direct approach to the study of inheritance systems. The notion of inheritance they consider in (the bulk of) that paper is essentially the same we have assumed here. The semantic problem is instead approached from a completely different perspective. Their view is strictly transformational. The methodology to capture the meaning of an inheritance system is to transform it into a logic program to then show the equivalence between the respective operational semantics. A declarative interpretation is then derived indirectly on the account of the well-known equivalence between the operational and declarative semantics in logic programming. A refined result is described in [25] where they introduce a direct declarative characterization for a composite language which combines the static and dynamic interpretations of inheritance as well as the overriding and extension *modes* between inherited definition we have considered in this paper. There is a fundamental difference from the approach we have presented here: the semantic construction of [25] applies to *complete* hierarchies and it is given under the assumption that the components of the hierarchy are known in advance. As such, the issue of compositionality is not taken into account.

Compositionality is instead one of the key issues in our approach: each differential program is looked at as an independent fragment to be arbitrarily composed onto any hierarchy. Then the compositional properties of our semantics ensure that the meaning of the resulting hierarchy can be determined from the meaning of the components.

In [7], Brogi et al. study a compositional semantics for a logic language equipped with mechanisms for message passing and inheritance. Their approach is rather differ-

ent than the one presented here, in at least two respects. The first is that our semantics is (cas)-correct whereas the semantics of [7] captures a less refined notion of operational behaviour stated in terms on the notion of (ground) success set. The second is that the definition of inheritance assumed in that paper is based only on the idea of *extension* rather than overriding between inherited definitions. This assumption is crucial in the definition of semantic framework presented in [7].

A compositional semantics of inheritance is also given in [8], but different semantic objects (the least Herbrand model and the immediate-consequence operator respectively) are required to coexist there, in order to capture the meaning of static and dynamic inheritance. In contrast to that case, the choice of context-sensitive interpretations, allows us to have a uniform treatment of the two mechanisms.

A modular extension to logic programming was also proposed by Sannella and Wallen in [30], based on the theory of modularity developed by the Standard ML module system. Abstraction and the ability to define structured components are also at the basis of that approach but cross-references between predicate definitions in different modules are achieved only through the explicit use of *qualified* names. Thus, there is no support for the implicit interaction between different components which is entailed by the composition mechanisms we have considered in this paper.

References

[1] K. R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.

[2] A. Bossi, M. Bugliesi, M. Gabbrielli, G. Levi, and M. C. Meo. Differential Logic Programs. In *Proc. 20th Annual ACM Symp. on Principles of Programming Languages*, pages 359–370. ACM Press, 1993.

[3] A. Bossi, M. Gabbrielli, G. Levi, and M. C. Meo. Contributions to the Semantics of Open Logic Programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, pages 570–580, 1992.

[4] A. Bossi, M. Gabbrielli, G. Levi, and M. C. Meo. A Compositional Semantics for Logic Programs. Technical Report, Dipartimento di Informatica, Universitá di Pisa, 1993. To appear in *Theoretical Computer Science*.

[5] A. Bossi and M. Menegus. Una Semantica Composizionale per Programmi Logici Aperti. In P. Asirelli, editor, *Proc. Sixth Italian Conference on Logic Programming*, pages 95–109, 1991.

[6] A. Brogi, E. Lamma, and P. Mello. Compositional Model-theoretic Semantics for Logic Programs. *New Generation Computing*, 11(1), 1992.

[7] A. Brogi, E. Lamma, and P. Mello. Objects in a Logic Programming Framework. In A.Voronkov, editor, *Logic Programming*, Lecture Notes in Artificial Intelligence, pages 102–113. Springer-Verlag, 1992.

[8] M. Bugliesi. A declarative view of inheritance in logic programming. In K. Apt, editor, *Proc. Joint Int. Conference and Symposium on Logic Programming*, pages 113–130. The MIT Press, 1992.

[9] M. Bugliesi, E. Lamma, and P. Mello. Modularity in Logic Programming. Technical Report 4/242, Progetto Finalizzato C.N.R. Sistemi Informatic e Calcolo Parallelo, 1993. (submitted for publication).

[10] W. Cook and J. Palsberg. A Denotational Semantics of Inheritance and its Correctness. In *Proceedings of OOPSLA'89*, pages 433–443. ACM, 1989.

[11] P. Cousot and R. Cousot. Abstract intepretation and application to logic programs. *Journal of Logic Programming*, 13(2&3):103-179, 1992.

[12] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative Modeling of the Operational Behavior of Logic Languages. *Theoretical Computer Science*, 69(3):289–318, 1989.

[13] M. Gabbrielli and G. Levi. On the Semantics of Logic Programs. In J. Leach Albert, B. Monien, and M. Rodriguez-Artalejo, editors, *Automata, Languages and Programming, 18th International Colloquium*, volume 510 of *LNCS*, pages 1–19. Springer-Verlag, 1991.

[14] H. Gaifman and E. Shapiro. Fully abstract compositional semantics for logic programs. In *Proc. 16th Annual ACM Symp. on Principles of Programming Languages*, pages 134–142. ACM, 1989.

[15] L. Giordano and A. Martelli and G.F. Rossi. Extending Horn Clause Logic with Modules Constructs. *Theoretical Computer Science*, 95:43-74, 1992.

[16] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[17] R. Kowalsky. *Logic for Problem Solving.* Elsevier North-Holland, 1979.

[18] J.-L. Lassez and M. J. Maher. Closures and Fairness in the Semantics of Programming Logic. *Theoretical Computer Science*, 29:167–184, 1984.

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

[20] J. W. Lloyd and J. C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242, 1991.

[21] P. Mancarella and D. Pedreschi. An Algebra of Logic Programs. In R. A. Kowalski and K. A. Bowen, editors, *Proc. 5th Int. Conference on Logic Programming*, pages 1006–1023. The MIT Press, 1988.

[22] D. Miller. A Logical Analysis of Modules in Logic Programming. *Journal of Logic Programming*, 6(2):79–108, 1989.

[23] L. Monteiro and A. Porto. Contextual Logic Programming. In G. Levi snd M. Martelli, editors, *Proc. of the 6th Int.l Conference on Logic Programming*, pages 284-299. The MIT Press, 1989.

[24] L. Monteiro and A. Porto. A Transformational View of Inheritance in Logic Programming. In D.H.D. Warren and P. Szeredi, editors, *Proc. 7th Int. Conference on Logic Programming*, pages 481–494. The MIT Press, 1990.

[25] L. Monteiro and A. Porto. Syntactic and Semantic Inheritance in Logic Programming. In J. Darlington and R. Dietrich, editors, *Proc. Workshop on Declarative Programming, Sasbachwalden 1991*, Workshops in Computing. Springer-Verlag, 1991.

[26] L. Monteiro and A. Porto. A Language for Contextual Logic Programming. In J.W. de Bakker K.R. Apt and J.J.M.M. Rutten, editors, *Logic Programming Languages, Constraints, Functions and Objects*. The MIT Press, 1993.

[27] Y. Moscowitz and E. Shapiro. Lexical Logic Programs. In K. Furukawa, editor, *Proc. of the 6th Int.l Conference on Logic Programming*, pages 2349-363. The MIT Press, 1991.

[28] R. O'Keefe. Towards an algebra for constructing logic programs. In J. Cohen and J. Conery, editors, *Proceedings of IEEE Symposium on Logic Programming*, pages 152–160. IEEE Computer Society Press, 1985.

[29] U. Reddy. Objects as Closures: Abstract Semantics of Object Oriented Languages. In *Proc. of the Int. Conf. on Lisp and Functional Programming*, pages 289–297. ACM, 1988.

[30] D. T. Sannella and L. A. Wallen. A Calculus for the Construction of Modular Polog Programs. *Journal of Logic Programming*, 6(12):147–177, 1992.

[31] W. F. Tichy. Design, Implementation, and Evaluation of a Revision Control System. In *Proc. of the 6th Int. Conf. on Software Engineering, IEEE*, Tokyo, 1982.

[32] P. Wegner. Dimensions of Object-Based Language Design. In *Proc. of the OOPSLA '87*, 1987.

[33] C. Zaniolo. Deductive Databases - Theory meets Practice. In *Proc. 2nd Int. Conf. on Extended Database Technology*, 1990.

A Appendix

In this appendix we prove lemmata 2.8, 3.16 and 3.17. In doing so, we will also state and prove a number of technical results.

Proof of Lemma 2.8

We start with lemma 2.8 whose proof needs the following

Lemma A.1 *For P and Q differential programs, let c be a clause of P and $P \lhd Q = \xi(P) \cup \phi(Q^\star)$ (definition 2.4). Then for all $q \in \pi(Q)$, if $q \in \Sigma_Q \cup \iota(Q)$, then:*

$$(a) \qquad def(\phi(q), \xi(P)) = \oslash$$
$$(b) \qquad \xi(c) \notin def(\phi(q), P \lhd Q)$$

Dually, if $p \in \kappa(P)$, $p \in \Sigma_P \cup \iota(P)$ and c is a clause of Q, then

$$(c) \qquad def(\xi(p), \phi(Q^\star)) = \oslash$$
$$(d) \qquad \phi(c) \notin def(\xi(p), P \lhd Q)$$

Proof. *We prove points (a) and (b), the proof of (c) and (d) is similar.*
(a): Assume, by contradiction that $def(\phi(q), \xi(P)) \neq \oslash$. Then, there exists $p_1 \in \kappa(P)$ such that $\xi(p_1) = \phi(q)$. From lemma 2.6 this implies that $p_1 = q$. Hence, being $q \in \Sigma_Q \cup \iota(Q)$, by compatibility, either $p_1 \in \Sigma_P$ or $p_1 \in \iota(P)$. Since $p_1 \in \kappa(P)$, in neither case $p_1 \in \Gamma'$ whereas in both cases $q \in \Gamma$. But then $\xi(p_1) = p_1$ whereas $\phi(q) \neq q$ and hence $\xi(p_1) \neq \phi(q)$, a contradiction.
(b): Let $c = H$:- $Body$ and $p = Pred(H)$. Then $p \in \kappa(P)$ and, by definition of ξ, $\xi(p) = p$. Hence $\xi(c) \in def(\phi(q), P \lhd Q)$ only if $p = \phi(q)$. But it is easy to show that assuming $p = \phi(q)$ leads to a contradiction. In fact either $q \neq \phi(q)$ or $q = \phi(q)$. In the first case $\phi(q) \notin \pi(P)$ and then $\phi(q) \neq p$. In the second case, $p = q$ and then, by compatibility and definition of ϕ we have $q \neq \phi(q)$ and hence $p \neq \phi(q)$. ■

Lemma 2.8 Let $HP = P_n$ *isa* P_{n-1} *isa* $\cdots$ *isa* P_1 and let $HP_\lhd$ be the corresponding differential program. Let $p \in \pi(P^j_{HP})$ for $j \leq n$. Then, for all $k \leq n$,

$$c \in P_k \ \&\ c \in select(p, P_j, HP) \iff c \in P^k_{HP} \ \&\ \psi^k_{HP}(c) \in def(\psi^j_{HP}(p), HP_\lhd)$$

Proof. The proof is by induction on the structure of the *isa* hierarchy. The base case, when HP is composed of the single program P_1, is trivial since $\psi^1_{P_1}$ is the identity renaming and there is no subsetting.

Let's then assume that the claim holds for any hierarchy $H = P_{n-1}$ *isa* $\cdots$ *isa* P_1 and consider the hierarchy $HP = P_n$ *isa* H. The corresponding differential program is $HP_\lhd = P_n \lhd H_\lhd = \xi_n(P_n) \cup \phi_{n-1}(H^\star_\lhd)$. We proceed distinguishing the cases when p is respectively a Σ, Δ, Θ or internal predicate.

Σ-*predicates* Let $p \in \Sigma_{P_j}$ be a predicate of $\pi(P^j_{HP})$, $j \leq n$. First consider the case when $j < n$. From the definition of the proof predicate $\vdash$, it follows that

$$select(p, P_j, HP) \quad = \quad select(p, P_j, H). \tag{1}$$

Correspondingly, we can show that in the differential program

$$def(\psi^j_{HP}(p), HP_\triangleleft) \quad = \quad \phi_{n-1}(def(\psi^j_H(p), H_\triangleleft)). \tag{2}$$

In fact, by definition, $def(\psi^j_{HP}(p), HP_\triangleleft) = def(\psi^j_{HP}(p), \xi_n(P_n)) \cup def(\psi^j_{HP}(p), \phi_{n-1}(H^\star_\triangleleft))$. By lemma A.1.(a) $def(\psi^j_{HP}(p), \xi_n(P_n)) = \oslash$ being $\psi^j_{HP}(p) = \phi_{n-1}(\psi^j_H(p))$. Furthermore, $def(\psi^j_{HP}(p), \phi_{n-1}(H^\star_\triangleleft)) = \phi_{n-1}(def(\psi^j_H(p), H^\star_\triangleleft)) = \phi_{n-1}(def(\psi^j_H(p), H_\triangleleft))$ because the subsetting doesn't erase the definition of any Σ-predicate.

We now notice that for $k = n$ the claim reduces to $false \iff false$. In fact, $c \in P_n$ implies by (1) that $c \notin select(p, P_j, HP)$. Correspondingly, in the differential program, $\psi^n_{HP} = \xi_n$ and $P^n_{HP} = P_n$. By lemma A.1.(b), $c \in P_n$ implies that $\xi_n(c) \notin def(\phi_{n-1}(\psi^j_H(p)), HP_\triangleleft)$. Hence, we may assume that $k < n$. Notice that for any clause c defining a Σ-predicate, $c \in P^k_{HP}$ if and only if $c \in P^k_H$. We use q as a shorthand for $\psi^j_H(p)$ and proceed with the following argument:

$$\begin{aligned}
c \in P_k \;\&\; c \in select(p, P_j, HP) &\overset{(1)}{\iff} c \in P_k \;\&\; c \in select(p, P_j, H) \\
\text{(by the inductive hyp)} &\iff c \in P^k_H \;\&\; \psi^k_H(c) \in def(q, H_\triangleleft) \\
(\phi_{n-1} \text{ is injective on } \pi(H_\triangleleft)) &\iff c \in P^k_H \;\&\; \phi_{n-1}(\psi^k_H(c)) \in \phi_{n-1}(def(q, H_\triangleleft)) \\
\text{(by (2))} &\iff c \in P^k_H \;\&\; \phi_{n-1}(\psi^k_H(c)) \in def(\phi_{n-1}(q), HP_\triangleleft) \\
(\text{since } c \text{ defines a } \Sigma\text{-predicate}) &\iff c \in P^k_{HP} \;\&\; \phi_{n-1}(\psi^k_H(c)) \in def(\phi_{n-1}(q), HP_\triangleleft) \\
(\text{since } \psi^k_{HP} = \psi^k_H \circ \phi_{n-1}). &\iff c \in P^k_{HP} \;\&\; \psi^k_{HP}(c) \in def(\phi_{n-1}(q), HP_\triangleleft)
\end{aligned}$$

Now consider the case when $j = n$. We distinguish two subcases. First assume that $p \in \kappa(P_n)$. If $k < n$ then we can show that the claim reduces again to $false \iff false$. The left side reduces to $false$, for $k < n$, because $select(p, P_n, HP) = def(p, P_n)$. As for the right side, for $k < n$, $\psi^k_{HP}(c) = \phi_{n-1}(\psi^k_H(c))$ and, by lemma A.1.(d), $\phi_{n-1}(\psi^k_H(c)) \notin def(\xi_n(p), HP_\triangleleft)$.

Let's then assume $k = n$. Since $P^n_{HP} = P_n$, obviously $c \in P_n \iff c \in P^n_{HP}$ and we can proceed as follows:

$$\begin{aligned}
c \in select(p, P_n, HP) &\iff c \in def(p, P_n) \\
&\iff \xi_n(c) \in \xi_n(def(p, P_n)) \\
&\iff \xi_n(c) \in def(\xi_n(p), HP_\triangleleft)
\end{aligned}$$

The last step follows from lemma A.1.(c), being $def(\xi_n(p), HP_\triangleleft) = def(\xi_n(p), \xi_n(P_n)) \cup def(\xi_n(p), \phi_{n-1}(H^\star_\triangleleft))$.

Finally consider the case when $p \notin \kappa(P_n)$. Assume that $p \in \kappa(P_i)$ for some component P_i of HP and let P_l the top-most among these components. Then for all m such

that $n \geq m > l, p \notin \kappa(P_m)$ and the definition of the proof predicate $\vdash$ implies that: $select(p, P_n, HP) = select(p, P_l, HP)$ Correspondingly, in the differential program we can show that:

$$def(\psi^l_{HP}(p), HP_\triangleleft) = def(\xi_n(p), HP_\triangleleft) \tag{3}$$

In fact, let $H_l = P_l \triangleleft \cdots \triangleleft P_1$, since p is a Σ-predicate in $\kappa(P_l)$ $p = \xi_l(p) = \psi^l_{H_l}(p)$. Since p is not defined in P_{l+1}, by definition, $p = \xi_{l+1}(p) = \phi_l(\psi^l_{H_l}(p))$ and $\phi_l(\psi^l_{H_l}(p)) = \psi^l_{H_{l+1}}(p)$. By iterating this argument $n - l$ times we obtain $p = \psi^l_{HP}(p)$. Similarly, we can show that $p = \psi^n_{HP}(p) \stackrel{def}{=} \xi_n(p)$ thus proving (3).

Now we can reason as in the case $j < n$ since $l < n$ and $p \in \pi(P^l_{HP})$, being $p \in \kappa(P_l) \cap \Sigma_{P_l}$.

Assume finally that for all i, $p \notin \kappa(P_i)$. Then obviously $select(p, P_n, HP) = \oslash$. Correspondingly, $def(\xi_n(p), HP_\triangleleft) = \oslash$. In fact, $def(\xi_n(p), \xi_n(P_n)) = \oslash$ being ξ_n injective on $\pi(P_n)$, and $def(\xi_n(p), \phi_{n-1}(H^\star_\triangleleft)) = \oslash$ being $p \notin \kappa(H_\triangleleft)$ and $\xi_n(p) \neq \phi_{n-1}(p)$ by lemma 2.6.

Δ-*predicates* When $p \in \Delta_{P_j}$, from the definition of the proof predicate $\vdash$, we have:

$$select(p, P_j, HP) \quad = \quad \begin{cases} select(p, P_j, H) & \text{if } p \notin \kappa(P_n) \\ def(p, P_n) & \text{otherwise} \end{cases}$$

Note that, when $p \notin \kappa(P_n)$ the above equality is well defined only if $j < n$. However, for $j = n$ it is immediate to see that $p \notin \kappa(P_n)$ implies that $select(p, P_n, HP) = select(p, P_{n-1}, HP)$. Hence, restricting to the case $j < n$ does not cause any loss of generality.

Consider then the case $p \notin \kappa(P_n)$. Since p is a Δ-predicate all the renamings used to construct $HP_\triangleleft$ are identities on p. We can thus establish the following equality:

$$def(p, HP_\triangleleft) \quad = \quad \phi_{n-1}(def(p, H_\triangleleft)) \tag{4}$$

In fact, by definition $def(p, HP_\triangleleft) = def(p, \xi_n(P_n)) \cup def(p, \phi_{n-1}(H^\star_\triangleleft))$. Since $p \in \Delta_{HP_\triangleleft}$, if $p \in \pi(P_n)$, then by compatibility we have that $p \in \Delta_{P_n}$. Hence $def(p, \xi_n(P_n)) = \oslash$ being $p \notin \kappa(P_n)$. Thus, to complete the proof we have to show that $def(p, \phi_{n-1}(H^\star_\triangleleft)) = \phi_{n-1}(def(p, H_\triangleleft))$. To see this observe that $p \in \Delta_{HP_\triangleleft}$ implies also that $p = \phi_{n-1}(p)$ and then $def(p, \phi_{n-1}(H^\star_\triangleleft)) = \phi_{n-1}(def(p, H^\star_\triangleleft)) = \phi_{n-1}(def(p, H_\triangleleft))$ where the last equality holds being $p \notin \kappa(P_n)$ and hence there is no subsetting for the clauses defining p in $H_\triangleleft$.

For $k = n$ the claim reduces to $false \iff false$. That the left side is equivalent to $false$ for $k = n$ follows by observing that $c \in P_n$ and $p \notin \kappa(P_n)$ implies that $Pred(head(c)) \neq p$ and hence $c \notin select(p, P_j, HP)$. As for the right side, $\psi^n_{HP} = \xi_n$ and $Pred(head(\xi_n(c))) \neq p$. Hence, $\xi_n(c) \notin def(p, HP_\triangleleft)$. We can thus restrict to the case $k < n$ and proceed similarly to the corresponding case of Σ-predicates. In fact,

$$c \in P_k \;\&\; c \in select(p, P_j, HP) \quad \iff \quad c \in P_k \;\&\; c \in select(p, P_j, H)$$

$$
\begin{array}{rcl}
(\text{by the inductive hyp}) & \iff & c \in P_H^k \ \&\ \psi_H^k(c) \in def(\psi_H^j(p), H_\lhd) \\
(\text{being } \psi_H^j(p) = p) & \iff & c \in P_H^k \ \&\ \psi_H^k(c) \in def(p, H_\lhd) \\
(\phi_{n-1} \text{ is injective on } \pi(H_\lhd)) & \iff & c \in P_H^k \ \&\ \phi_{n-1}(\psi_H^k(c)) \in \phi_{n-1}(def(p, H_\lhd)) \\
(\text{by (4)}) & \iff & c \in P_H^k \ \&\ \phi_{n-1}(\psi_H^k(c)) \in def(p, HP_\lhd) \\
(\text{since } p \notin \kappa(P_n)) & \iff & c \in P_{HP}^k \ \&\ \phi_{n-1}(\psi_H^k(c)) \in def(p, HP_\lhd) \\
(\text{being } \psi_{HP}^j(p) = p) & \iff & c \in P_{HP}^k \ \&\ \psi_{HP}^k(c) \in def(\psi_{HP}^j(p), HP_\lhd)
\end{array}
$$

Consider the case when $p \in \kappa(P_n)$. If $k < n$ then we can show that the claim reduces again to $false \iff false$. In fact $c \in select(p, P_j, HP)$ and $p \in \kappa(P_n)$ implies, by the definition of $\vdash$, that $c \notin P_k$. Correspondingly, $c \notin P_{HP}^k$ due to the subsetting. We can then restrict to the case $k = n$. Being $c \in P_{HP}^n \iff c \in P_n$, we proceed as follows:

$$
\begin{array}{rcl}
c \in select(p, P_j, HP) & \iff & c \in def(p, P_n) \\
(\text{by injectivity of } \xi_n \stackrel{def}{=} \psi_{HP}^n) & \iff & \xi_n(c) \in \xi_n(def(p, P_n)) \\
(\text{since } \xi_n(p) = \psi_{HP}^j(p) = p) & \iff & \xi_n(c) \in def(\psi_{HP}^j(p), \xi_n(P_n)) \\
& \iff & \psi_{HP}^n(c) \in def(\psi_{HP}^j(p), HP_\lhd).
\end{array}
$$

The last step follows being $def(p, HP_\lhd) = def(p, \xi_n(P_n))$ since $def(p, \phi_{n-1}(H_\lhd^\star)) = \oslash$.

Θ-*predicates.* When $p \in \Theta_{P_j}$, from the definition of the proof predicate $\vdash$, we have:

$$
select(p, P_j, HP) \ = \ \bigcup_{i \le n} def(p, P_i)
$$

Since p is a Θ-predicate all the renamings used to construct $HP_\lhd$ are identities on p. Furthermore, for all k if c is a clause defining a Θ-predicate then $c \in P_k \iff c \in P_{HP}^k$. Hence, for the differential program $HP_\lhd$,

$$
def(p, HP_\lhd) \ = \ \bigcup_{i \le n} def(p, \psi_{HP}^i(P_{HP}^i)) \ = \ \bigcup_{i \le n} \psi_{HP}^i(def(p, P_i))
$$

Finally:

$$
\begin{array}{rcl}
c \in select(p, P_j, HP) & \iff & c \in def(p, P_k) \\
& \iff & \psi_{HP}^k(c) \in \psi_{HP}^k(def(p, P_k)) \\
& \iff & \psi_{HP}^k(c) \in def(p, HP_\lhd)
\end{array}
$$

Internal predicates. The proof is similar to the case of Σ-predicates. ■

Proof of Lemma 3.16

Lemma 3.16 can be proved by using the same arguments used to prove the compositionality of the open-semantics in [4]. Following the guidelines of [4] we then first introduce an unfolding semantics which we will prove equivalent to the fixed point semantics $[\![\cdot]\!]$.

Definition A.2 *Let Ω be a set of predicates. By $\mathcal{C}^{\Omega}$ we denote the set of all the (equivalence classes of) cs-clauses A :- $s \Box \tilde{B}$ such that $Pred(\tilde{B}) \subseteq \Omega$.*

Definition A.3 (Unfolding semantics) *Let P be a $\langle \Sigma, \Delta, \Theta \rangle$-program. Then we define the collection of $\langle \Sigma, \Delta, \Theta \rangle$-programs*

$$P_1 = P$$
$$P_{n+1} = unf_{P_n, \Omega(P), \Delta}(P \cup Id_{\Omega(P)}).$$

The unfolding semantics $\mathcal{U}(P)$ of the program P is defined as

$$\mathcal{U}(P) = \alpha(\bigcup_{n=1,2,\ldots} P_n \cap \mathcal{C}^{\Omega(P)}) \ \setminus \ \mathcal{D}(\iota(P)).$$

where $\mathcal{D}(\iota(P))$ is defined as in definition 3.9.

In order to prove the equivalence of $\mathcal{U}(P)$ and $[\![P]\!]$ we need two lemmata (A.5 and A.6 below). The first one states a weak form of associativity for the cs-unfolding operator. The second shows the equivalence between the intermediate steps in the construction of the fixpoint and of the unfolding semantics.

By $mgu(E)$ we denote the set of idempotent most general unifier of E. It is well known that the idempotent mgu is unique up to renaming. Then, in the following, we will use also the notation $mgu(E) = \vartheta$ to mean that ϑ is an (unique up to renaming) mgu of E.

Lemma A.4 *[3] Let E_1, E_2 be sets of equations with $mgu(E_1) = \vartheta$ and $mgu(E_2\vartheta) = \gamma$. Then $mgu(E_1 \cup E_2) = \vartheta\gamma$.*

Lemma A.5 *Let P be a $\langle \Sigma, \Delta, \Theta \rangle$-program, Ψ be a set of predicate symbols and let Q, W be sets of cs-clauses which contain Id_{Ψ}. Then:*

$$unf_{P,\Psi,\Delta}(unf_{Q,\Psi,\Delta}(W)) =_{\alpha} unf_{unf_{P,\Psi,\Delta}(Q),\Psi,\Delta}(W)$$

Proof. *To simplify the notation, in the following we will use unf_P as a shorthand for $unf_{P,\Psi,\Delta}$ with the understanding that the subscripts Ψ, Δ will be the same for any P. Accordingly, we will prove that*

$$unf_P(unf_Q(W)) =_{\alpha} unf_{unf_P(Q)}(W)$$

By definition 3.6, $c \in unf_{unf_P(Q)}(W)$ iff the following conditions hold

1. $\exists\ H$:- $s \Box A_1, \ldots, A_n \in P$,
   for $i = 1, \ldots, n$, $\exists\ A'_i$:- $s_i \Box B_{i,1}, \ldots, B_{i,m_i} \in Q$,
   for $j = 1, \ldots, m_i$, $\exists\ B'_{i,j}$:- $s'_{i,j} \Box \tilde{C}_{i,j} \in W$
2. $\exists\ \vartheta = mgu(\{A_i = A'_i\}_{i=1,\ldots,n})$,
   $\exists\ \gamma = mgu(\{B_{i,1}\vartheta = B'_{i,1}, \ldots, B_{i,m_i}\vartheta = B'_{i,m_i}\}_{i=1,\ldots,n})$,

3. $c = H\vartheta\gamma \text{:-} s_c \Box(\tilde{C}_{1,1}, \ldots, \tilde{C}_{n,m_n})\gamma$,

where

$$s_c = s \cup (\bigcup_i s_i) \cup (\bigcup_{i,j} s'_{i,j}) \cup \mathcal{S}_A \cup \mathcal{S}_B$$

and

$\mathcal{S}_A = \bigcup_{i=1,\ldots,n} \mathcal{S}_{A_i}$
$\mathcal{S}_{A_i} = \{Pred(A'_i) \mid Pred(A'_i) \in \Delta \text{ and } A'_i \text{:-} s_i \Box B_{i,1}, \ldots, B_{i,m_i} \notin Id_\Psi\}$
$\mathcal{S}_B = \bigcup_{i=1,\ldots,n} \mathcal{S}_{B_i}$
$\mathcal{S}_{B_i} = \{Pred(B'_{i,j}) \mid Pred(B'_{i,j}) \in \Delta \text{ and } B'_{i,j} \text{:-} s'_{i,j} \Box \tilde{C}_{i,j} \notin Id_\Psi,\ j = 1, \ldots, m_i\}$

On the other side, $c' \in unf_P(unf_Q(W))$ *iff*

1'. $\exists\ H \text{:-} s \Box A_1, \ldots, A_n \in P$,
for $i = 1, \ldots, n$, $\exists\ A'_i \text{:-} s_i \Box B_{i,1}, \ldots, B_{i,m_i} \in Q$,
for $j = 1, \ldots, m_i$, $\exists\ B'_{i,j} \text{:-} s'_{i,j} \Box \tilde{C}_{i,j} \in W$,

2'. $\exists\ \beta = mgu(\{B_{i,1} = B'_{i,1}, \ldots, B_{i,m_i} = B'_{i,m_i}\}_{i=1,\ldots,n})$,
$\exists\ \delta = mgu(\{A_i = A'_i\beta\}_{i=1,\ldots,n})$,

3'. $c' = H\delta \text{:-} s_{c'} \Box(\tilde{C}_{1,1}, \ldots, \tilde{C}_{n,m_n})\beta\delta$,

where

$$s_{c'} = s \cup (\bigcup_i s_i) \cup (\bigcup_{i,j} s'_{i,j}) \cup \mathcal{S}'_A \cup \mathcal{S}_B$$

$\mathcal{S}_B$ *is defined as above whereas* $\mathcal{S}'_A$ *is defined below:*

$\mathcal{S}'_A = \bigcup_{i=1,\ldots,n} \mathcal{S}'_{A_i}$
$\mathcal{S}'_{A_i} = \{Pred(A'_i) \mid Pred(A'_i) \in \Delta \text{ and } A'_i \text{:-} s_i \cup \mathcal{S}_{B_i} \cup \bigcup_j s'_{i,j} \Box \tilde{C}_{i,1}, \ldots, \tilde{C}_{i,m_i} \notin Id_\Psi\}$

Note that in 2'. we can use a unique β *instead of* n $\beta_i = mgu(B_{i,1} = B'_{i,1}, \ldots, B_{i,m_i} = B'_{i,m_i})$ *for* $i = 1, \ldots, n$ *because the clauses of* W *are renamed apart.*

What we can show by now is that the head and the body of c *and* c' *are equal up to renaming. In fact, by lemma A.4,* $\vartheta\gamma = mgu(\{A_i = A'_i\}_{i=1,\ldots,n} \cup \{B_{i,1} = B'_{i,1}, \ldots, B_{i,m_i}\}_{i=1,\ldots,n})$, *and, since the clauses of* W *are rename apart,* $\vartheta\gamma = \beta\delta$ *up to renaming. Furthermore,* $(\tilde{C}_{1,1}, \ldots, \tilde{C}_{n,m_n})\gamma = (\tilde{C}_{1,1}, \ldots, \tilde{C}_{n,m_n})\vartheta\gamma$ *because the clauses of* W *are renamed apart and hence we can choose them so that the* $\tilde{C}_{i,j}$ *have no variable in the domain of* ϑ *(whereby* $\tilde{C}_{i,j}\vartheta = \tilde{C}_{i,j}$*). For the same reason,* $H\delta = H\beta\delta$ *and hence the claim.*

The problem is that, in general $s_{c'} \neq s_c$ *being* $\mathcal{S}'_A \neq \mathcal{S}_A$*. However, we can show that there exists a cs-clause* $c'' \in unf_{unf_P(Q)}(W) \cap unf_P(unf_Q(W))$, *which has the same head and body as* c' *(and* c*), and whose set of constraints* $s_{c''}$ *is contained or equal to* $s_c \cap s_{c'}$*. This, by the definition of the abstraction* α*, ensures that* $\alpha(unf_{unf_P(Q)}(W)) = \alpha(unf_P(unf_Q(W)))$*. We proceed as follows. Being* $\mathcal{S}'_A \neq \mathcal{S}_A$*, there must exist* k *such*

that $\mathcal{S}'_{A_k} \neq \mathcal{S}_{A_k}$*, i.e. such that either* $Pred(A_k) \notin \mathcal{S}'_{A_k}$ *and* $Pred(A_k) \in \mathcal{S}_{A_k}$*, or, dually,* $Pred(A_k) \in \mathcal{S}'_{A_k}$ *and* $Pred(A_k) \notin \mathcal{S}_{A_k}$*. For any such* k*, let*

$$cl_1 = A'_k \text{:-} s_k \Box B_{k,1}, \ldots, B_{k,m_k} \in Q \text{ and}$$
$$cl_2 = A'_k \text{:-} s_k \cup \mathcal{S}_{B_i} \cup \bigcup_j s'_{k,j} \Box \tilde{C}_{k,1}, \ldots, \tilde{C}_{k,m_k} \in unf_Q(W).$$

be the clauses used to unfold A_k *in* $H \text{:-} s \Box A_1, \ldots A_k$ *to produce respectively* c *and* c'*. Now consider the two cases separately.*

1. $Pred(A_k) \notin \mathcal{S}'_{A_k}$ *and* $Pred(A_k) \in \mathcal{S}_{A_k}$*. From* $Pred(A_k) \notin \mathcal{S}'_{A_k}$*, it follows that* $cl_2 \in Id_\Psi$ *and hence* $cl_2 = A'_k \text{:-} A'_k$*. Now, since both* Q *and* W *contain* Id_Ψ*, we can use* $A'_k \text{:-} A'_k \in Id_\Psi$ *to unfold twice* A_k *in* $H \text{:-} s \Box A_1, \ldots, A_n \in P$*. But then* $unf_{unf_P(Q)}(W)$ *contains a cs-clause which has the same head and body as* c*, and whose (sub)set of constraints* $\mathcal{S}_{A_k}$ *does not contain* $Pred(A_k)$*.*

2. $Pred(A_k) \in \mathcal{S}'_{A_k}$ *and* $Pred(A_k) \notin \mathcal{S}_{A_k}$*. Now, from* $Pred(A_k) \notin \mathcal{S}_{A_k}$ *it follows that* $cl_1 \in Id_\Psi$*, i.e.* $cl_1 = A'_k \text{:-} A'_k$*. But then, since* W *contains* Id_Ψ*, we can again choose* $A'_k \text{:-} A'_k$ *to unfold* A_k *in* cl_2 *and use the resulting clause (which is a clause in* $unf_Q(W)$*), to unfold* $H \text{:-} s \Box A_1, \ldots, A_n \in P$*. From the latter unfolding step we obtain a clause in* $unf_P(unf_Q(W))$ *which has the same head and body as* c' *and which does not contain* $Pred(A_k)$ *in its (sub)set of constraints* $\mathcal{S}'_{A_k}$*.*

The existence of $c'' \in unf_{unf_P(Q)}(W) \cap unf_P(unf_Q(W))$*, with the expected properties follows now immediately being the choice of* $Pred(A_k)$ *arbitrary.* ■

Lemma A.6 *Let* P *be a* $\langle \Sigma, \Delta, \Theta \rangle$*-program, let* P_n *be as in definition A.3. Then*

$$T_P^{cs} \uparrow n =_\alpha P_n \cap \mathcal{C}^{\Omega(P)}.$$

Proof. *In the following, given a set of cs-clauses* W*, we use the notation*

$$unf_P^1(W) = unf_{P,\Omega(P),\Delta}(W) \text{ and, for } n > 1,$$
$$unf_P^n(W) = unf_{P,\Omega(P),\Delta}(unf_P^{n-1}(W)).$$

Similarly to what we have done before, to simplify the notation, we will abbreviate $unf_{P_i,\Omega(P),\Delta}$ *in* unf_{P_i} *(the subscripts* $\Omega(P), \Delta$ *will be the same for any* P_i*). Before proving the thesis we need three properties of the unfolding. First note that, by a straightforward induction on* n*, it can be proved that, for* $n \geq 1$*,*

$$unf_P(unf_P^{n-1}(W)) = unf_P^{n-1}(unf_P(W)) \tag{1}$$

Assume now that W *contains* $Id_{\Omega(P)}$*. Then for* $n \geq 1$ *the following equivalence holds*

$$unf_{P_{n+1}}(W) =_\alpha unf_P(unf^n_{P \cup Id_{\Omega(P)}}(W)) \tag{2}$$

The proof is by induction on n.
For $(n = 1)$ we have

$$\begin{array}{lll} unf_{P_2}(W) & = & \textit{(by definition A.3)} \\ unf_{unf_P(P \cup Id_{\Omega(P)})}(W) & =_\alpha & \textit{(by lemma A.5)} \\ unf_P(unf_{P \cup Id_{\Omega(P)}}(W)) & & \end{array}$$

For $(n > 1)$ assume that $unf_{P_n}(W) =_\alpha unf_P(unf^{n-1}_{P \cup Id_{\Omega(P)}}(W))$. Then

$$\begin{array}{lll} unf_{P_{n+1}}(W) & = & \textit{(by definition A.3)} \\ unf_{unf_{P_n}(P \cup Id_{\Omega(P)})}(W) & =_\alpha & \textit{(by lemma A.5)} \\ unf_{P_n}(unf_{P \cup Id_{\Omega(P)}}(W)) & =_\alpha & \textit{(by the inductive hypothesis)} \\ unf_P(unf^{n-1}_{P \cup Id_{\Omega(P)}}(unf_{P \cup Id_{\Omega(P)}}(W))) & = & \textit{(by definition of } unf^n_P \textit{ and (1))} \\ unf_P(unf^n_{P \cup Id_{\Omega(P)}}(W)) & & \end{array}$$

We can apply the inductive hypothesis in the previous step, because if W contains $Id_{\Omega(P)}$ then so does $unf_{P \cup Id_{\Omega(P)}}(W)$. This concludes the proof of (2).
Now notice that, by definition of $\uparrow$ and by definition 3.6,

$$T^{cs}_P \uparrow n = unf_P(Id_{\Omega(P)} \cup T^{cs}_P \uparrow n - 1) \tag{3}$$

Finally we show that for $n \geq 1$

$$unf^n_{P \cup Id_{\Omega(P)}}(Id_{\Omega(P)}) = Id_{\Omega(P)} \cup (T^{cs}_P \uparrow n). \tag{4}$$

Also in this case the proof is by induction on n. For $(n = 1)$ we have the following equivalences

$$\begin{array}{lll} unf_{P \cup Id_{\Omega(P)}}(Id_{\Omega(P)}) & = & \textit{(by definition of unfolding)} \\ unf_P(Id_{\Omega(P)}) \cup unf_{Id_{\Omega(P)}}(Id_{\Omega(P)}) & = & \textit{(by definition of } Id_{\Omega(P)}\textit{)} \\ unf_P(Id_{\Omega(P)}) \cup Id_{\Omega(P)} & = & \textit{(by (3))} \\ (T^{cs}_P \uparrow 1) \cup Id_{\Omega(P)} & & \end{array}$$

For $(n > 1)$ assume $unf^n_{P \cup Id_{\Omega(P)}}(Id_{\Omega(P)}) = Id_{\Omega(P)} \cup T^{cs}_P \uparrow n$.

$$\begin{array}{lll} unf^{n+1}_{P \cup Id_{\Omega(P)}}(Id_{\Omega(P)}) & = & \textit{(by definition of } unf^{n+1}\textit{)} \\ unf_{P \cup Id_{\Omega(P)}}(unf^n_{P \cup Id_{\Omega(P)}}(Id_{\Omega(P)})) & = & \textit{(by inductive hypothesis)} \\ unf_{P \cup Id_{\Omega(P)}}(Id_{\Omega(P)} \cup T^{cs}_P \uparrow n) & = & \textit{(by definition of unfolding)} \\ unf_P(Id_{\Omega(P)} \cup T^{cs}_P \uparrow n) \cup unf_{Id_{\Omega(P)}}(Id_{\Omega(P)} \cup T^{cs}_P \uparrow n) & = & \textit{(by definition of } Id_{\Omega(P)}\textit{)} \\ unf_P(Id_{\Omega(P)} \cup T^{cs}_P \uparrow n) \cup unf_{Id_{\Omega(P)}}(T^{cs}_P \uparrow n) \cup Id_{\Omega(P)} & = & \textit{(by (3))} \\ T^{cs}_P \uparrow n+1 \cup Id_{\Omega(P)} \cup unf_{Id_{\Omega(P)}}(T^{cs}_P \uparrow n) & = & \textit{(by the following remark)} \\ T^{cs}_P \uparrow n+1 \cup Id_{\Omega(P)} & & \end{array}$$

The last equality holds because, by definition of $Id_{\Omega(P)}$ and of the unfolding rule, $unf_{Id_{\Omega(P)}}(T^{cs}_P \uparrow n) \subseteq T^{cs}_P \uparrow n$ and since T^{cs}_P is monotonic, $T^{cs}_P \uparrow n \subseteq T^{cs}_P \uparrow n+1$. This completes the proof of (4).

We can now prove the thesis of the lemma. Note that, from definition A.3, we have that:

$$P_n \cap \mathcal{C}^{\Omega(P)} = unf_{P_n}(Id_{\Omega(P)}) = T^{cs}_{P_n}(\emptyset) \tag{5}$$

Then, for $n = 1$ we have obviously $P_1 \cap \mathcal{C}^{\Omega(P)} = T^{cs}_P \uparrow 1$ (recall that $P_1 = P$).
For $n > 1$ we have the following equivalences

$$\begin{array}{lll} T^{cs}_P \uparrow n & = & \textit{(by 3)} \\ unf_P(Id_{\Omega(P)} \cup T^{cs}_P \uparrow n-1) & = & \textit{(by 4)} \\ unf_P(unf^{n-1}_{P \cup Id_{\Omega(P)}}(Id_{\Omega(P)})) & =_\alpha & \textit{(by 2)} \\ unf_{P_n}(Id_{\Omega(P)}) & = & \textit{(by 5)} \\ P_n \cap \mathcal{C}^{\Omega(P)} & & \end{array}$$

and this completes the proof. ■

We can now state the following

Theorem A.7 *Let P be a $\langle \Sigma, \Delta, \Theta \rangle$-program. Then $[\![P]\!] = \mathcal{U}(P)$.*

Proof Immediate from lemma A.6.

■

We are now ready to prove lemma 3.16.

Lemma 3.16 Let $\xi(P)$ and $\phi(Q^\star)$ be defined according to definition 2.4. Then

$$[\![\xi(P) \cup \phi(Q^\star)]\!] \quad = \quad [\![[\![\xi(P)]\!] \cup [\![\phi(Q^\star)]\!]]\!]$$

Proof sketch. Following the guidelines of [4], this result is proved by first introducing an operational semantics which is (1) compositional and (2) equal to the unfolding semantics, and then using previous theorem A.7. The proofs of (1) and (2) use exactly the same arguments used to prove theorem 2.13 (previously shown as theorem 3.2) and theorem 4.14 in in [4] respectively. Therefore here we point out the modification needed to adapt those proofs to our case. The reader is referred to [4] for fuller details.

Firstly, note that the meaning of the set Ω for an Ω-open program P in [3, 4] (and as discussed in section 3) is exactly the same as that one of the set $\Omega(P)$ of open predicates for a $\langle \Sigma, \Delta, \Theta \rangle$-program P. Then we can obtain an "open" operational semantics for differential programs by using the unfolding rule as follows. Le us consider resolvents of the form $s \Box B_1, \ldots, B_n$, and given a selection rule R, let us define a derivation step $\stackrel{\vartheta}{\leadsto}{}^{cs}{}_{P,R}$ in the $\langle \Sigma, \Delta, \Theta \rangle$-program P as

$$s \Box A_1, \ldots, A_i, \ldots A_n \stackrel{\vartheta}{\leadsto}{}^{cs}{}_{P,R} (s \cup s' \Box A_1, \ldots, B_1, \ldots, B_m, \ldots A_n)\vartheta$$

if and only if $(A_i \text{:-} s' \Box B_1, \ldots, B_m)\vartheta \in unf^{cs}_{Id, Open(P), \Delta}(P)$ where $Id = \{A_i \text{:-} A_i\}$ and A_i is the atom selected by R in $A_1, \ldots, A_n$.

The open operational semantics $\mathcal{O}(P)$ is obtained by repeating the construction of definition 2.5 in [4] (shown in in section 3). Formally we can define

$$\mathcal{O}'(P) = \{p(\tilde{X})\vartheta :\!- s\Box\tilde{B} \mid \quad \exists\ R\ s.t.\ p(\tilde{X}) \overset{\vartheta}{\rightsquigarrow}{}^{cs}{}_{P,R}\ s\Box\tilde{B}, Pred(\tilde{B}) \subseteq \Omega(P)\}$$

$$\mathcal{O}(P) = \mathcal{O}'(P) \setminus \mathcal{D}(\iota(P))$$

where, by abusing notation, $\overset{\vartheta}{\rightsquigarrow}{}^{cs}{}_{P,R}$ denotes also a sequence of derivation steps.

Now we can repeat exactly the same proof of theorem 2.13 of [4] (previously shown as theorem 3.2) to show the OR-compositionality of $\mathcal{O}'(P)$. Namely we have that if (i) $\Omega(P \cup Q) \subseteq \Omega(P) \cup \Omega(Q)$ and (ii) $\pi(P) \cap \pi(Q) \subseteq \Omega(P) \cap \Omega(Q)$, then

$$\mathcal{O}'(P \cup Q) = \mathcal{O}'(\mathcal{O}'(P) \cup \mathcal{O}'(Q)) \tag{1}$$

By definition the bodies of the cs-clauses in $\mathcal{O}'(P)$ do not contain internal predicates. Moreover, clearly if W is a set of cs-clauses which do not contain internal predicates in the bodies we have

$$\mathcal{O}'\left(W \setminus \mathcal{D}(\iota(W))\right) = \mathcal{O}'(W) \setminus \mathcal{D}(\iota(W))$$

since the clauses defining internal predicates cannot be used in rewrite any atom in (the bodies of cluses in) W. Therefore from 1 we have

$$\mathcal{O}(P \cup Q) = \mathcal{O}(\mathcal{O}(P) \cup \mathcal{O}(Q)) \tag{2}$$

Now observe that, by definition 2.4, for the program $\xi(P) \cup \phi(Q^\star)$ we have

$$\begin{array}{l}\Omega(\xi(P) \cup \phi(Q^\star)) \subseteq \Omega(\xi(P)) \cup \Omega(\phi(Q^\star)) \text{ and} \\ \pi(\xi(P)) \cap \pi(\phi(Q^\star)) \subseteq \Omega(\xi(P)) \cap \Omega(\phi(Q^\star)).\end{array}$$

Therefore, from 2 we have

$$\mathcal{O}\left(\xi(P) \cup \phi(Q^\star)\right) = \mathcal{O}\left(\mathcal{O}(\xi(P)) \cup \mathcal{O}(\phi(Q^\star))\right). \tag{3}$$

Let us denote by $\mathcal{U}'(P)$ the unfolding semantics obtained from definition A.3 by not performing any deletion of clauses, i.e.

$$\mathcal{U}'(P) = \bigcup_{n=1,2,\ldots} P_n \cap \mathcal{C}^{\Omega(P)}.$$

The equality $\mathcal{O}'(P) = \mathcal{U}'(P)$ can be shown again by using exactly the same proof of theorem 4.14 in [4]. Therefore we have

$$\mathcal{U}(P) = \alpha(\mathcal{O}'(P)) \setminus \mathcal{D}(\iota(P)) = \alpha(\mathcal{O}(P)) \tag{4}$$

The last equality holds because, for any set of cs-clauses A and for any set of predicates Ψ, by definition 3.8 and by definition of $\mathcal{D}(\Psi)$ (the set of clauses whose head predicate

is in Ψ) we have $\alpha(A) \setminus \mathcal{D}(\Psi) = \alpha(A \setminus \mathcal{D}(\Psi))$. Moreover observe that, again by definition of α, for any A, B sets of cs-clauses

$$\alpha(A \cup B) = \alpha(\alpha(A) \cup \alpha(B)) \tag{5}$$

holds. Therefore, from 3, 4 and 5 we have

$$\mathcal{U}\left(\xi(P) \cup \phi(Q^\star)\right) = \mathcal{U}\left(\mathcal{U}(\xi(P)) \cup \mathcal{U}(\phi(Q^\star))\right) \tag{6}$$

and then the thesis follows from theorem A.7. ■

Proof of lemma 3.17

Again, we need the following intermediate result.

Lemma A.8 *Let $\langle \Sigma_P, \Delta_P, \Theta_P \rangle$-$P$ and $\langle \Sigma_Q, \Delta_Q, \Theta_Q \rangle$-$Q$ be differential programs and let $P \lhd Q$ be the differential program $\langle \Sigma, \Delta, \Theta \rangle$-$(\xi(P) \cup \phi(Q^\star))$ as defined in definition 2.4. Then $\forall n \geq 0$,*

1. $T^{cs}_{\xi(P)} \uparrow n = T^{cs}_P \uparrow n \setminus R$ *where* $R = \{H \text{:-} s \ \Box \ \tilde{B} \in \mathcal{C}_\Delta \mid Pred(\tilde{B}) \cap \Sigma_P \not\subseteq \kappa(Q)\}$
2. $T^{cs}_{\phi(Q^\star)} \uparrow n = \phi(T^{cs}_Q \uparrow n \setminus S)$ *where* $S = \{H \text{:-} s \ \Box \ \tilde{B} \in \mathcal{C}_\Delta \mid s \cap \kappa(P) \neq \emptyset$ *or* $Pred(H) \in \Delta_Q \cap \kappa(P)$ *or* $Pred(\tilde{B}) \cap \Sigma_Q \neq \emptyset\}$

Proof. *We prove 1. and 2. separately.*

Proof of 1. *The proof is by induction on n. The base case, for n=0 is immediate since* $T^{cs}_{\xi(P)} \uparrow 0 = \emptyset = T^{cs}_P \uparrow 0 \setminus R$.

Consider then the inductive case. By definition of ξ, $\Omega(\xi(P)) = \Omega(P) \setminus (\Sigma_P \setminus \kappa(Q))$. By this equality it follows that if $p \in \Sigma_P \cap \Omega(\xi(P))$ then p is defined in Q. Hence $p(\tilde{X})\text{:-}p(\tilde{X}) \notin R$ and therefore $Id_{\Omega(\xi(P))} = Id_{\Omega(P)} \setminus R$. Moreover note that $\Delta_P = \Delta_{\xi(P)}$ and that, for $q \in \Delta_P$, $q(\tilde{X})\text{:-}q(\tilde{X}) \in Id_{\Omega(P)}$ iff $q(\tilde{X})\text{:-}q(\tilde{X}) \in Id_{\Omega(\xi(P))}$. By these observations and by definition 3.6 we have

$$unf_{\xi(P),\Omega(\xi(P)),\Delta_{\xi(P)}}(I \cup Id_{\Omega(\xi(P))}) \quad = \quad unf_{\xi(P),\Omega(P),\Delta_P}(I \cup (Id_{\Omega(P)} \setminus R)). \tag{1}$$

Now, by definition 2.4, for any $c \in P$, if $\xi(c) \neq c$ then the body of c contains a predicate $p \in \Sigma_P$ which is not defined in $P \cup Q$. Then, by an obvious inductive argument, it can be shown that for any n no clause in $T^{cs}_P \uparrow n$ defines p. Similarly, by definition of R, no clause in $(Id_{\Omega(P)} \setminus R)$ defines p. Thus we have:

$$unf_{\xi(P),\Omega(P),\Delta_P}((T^{cs}_P \uparrow n \cup Id_{\Omega(P)}) \setminus R) \quad = \quad unf_{P,\Omega(P),\Delta_P}((T^{cs}_P \uparrow n \cup Id_{\Omega(P)}) \setminus R) \tag{2}$$

Then, to prove point 1. we proceed as follows.

$T^{cs}_{\xi(P)} \uparrow n$	=	*by definition of T^{cs} and by (1)*
$unf_{\xi(P),\Omega(P),\Delta_P}(T^{cs}_{\xi(P)} \uparrow n-1 \cup (Id_{\Omega(P)} \setminus R))$	=	*by inductive hypothesis*
$unf_{\xi(P),\Omega(P),\Delta_P}((T^{cs}_P \uparrow n-1 \cup Id_{\Omega(P)}) \setminus R)$	=	*by (2)*
$unf_{P,\Omega(P),\Delta_P}((T^{cs}_P \uparrow n-1 \cup Id_{\Omega(P)}) \setminus R)$	=	*by definition of unf and R*
$unf_{P,\Omega(P),\Delta_P}(T^{cs}_P \uparrow n-1 \cup Id_{\Omega(P)}) \setminus R$	=	*by definition of T^{cs}*
$T^{cs}_P \uparrow n \setminus R$.		

Proof of 2. *The proof is again by induction on n. As for 1., the base case is immediate since $T^{cs}_{\phi(Q^\star)} \uparrow 0 = \emptyset = \phi(T^{cs}_Q \uparrow 0 \setminus S)$. Consider then the inductive case. Observe that since dynamic predicate symbols are not renamed by ϕ, $\Delta_{\phi(Q^\star)} = \Delta_{Q^\star} = \Delta_Q \cap \pi(Q^\star)$. Therefore for $\Pi \subseteq \pi(Q^\star)$, $\Pi \cap \Delta_Q = \Pi \cap \Delta_{\phi(Q^\star)}$. For the same reason, for $\Pi \subseteq \pi(Q^\star)$, $\Pi \cap \Omega(Q) \cap \Delta_Q = \Pi \cap \Omega(Q^\star) \cap \Delta_{\phi(Q^\star)}$. Hence, from the definition of unfolding, if $c' \in Q^\star$ we have that:*

$$unf_{\{c'\},\Omega(Q),\Delta_Q}(I) \quad = \quad unf_{\{c'\},\Omega(\phi(Q^\star)),\Delta_{\phi(Q^\star)}}(I). \tag{1}$$

Now we prove the two inclusions separately.
$\left(T^{cs}_{\phi(Q^\star)} \uparrow n \subseteq \phi(T^{cs}_Q \uparrow n \setminus S)\right)$.
Assume that

$$cl = \phi(A \text{:-} s \ \Box \ \tilde{L}_1, \ldots, \tilde{L}_k)\vartheta \in T^{cs}_{\phi(Q^\star)} \uparrow n.$$

Note that, being ϕ injective, for any clause c', $c' \in Q^\star$ iff $\phi(c') \in \phi(Q^\star)$. Then by definition of $T^{cs}_{\phi(Q^\star)}$, it follows that there exist

$$\begin{array}{l} c' = A \text{:-} B_1, \ldots, B_k \in Q^\star, \\ c_i = \phi(B'_i \text{:-} s_i \ \Box \ \tilde{L}_i) \in T^{cs}_{\phi(Q^\star)} \uparrow n-1 \cup Id_{\Omega(\phi(Q^\star))}, \ i = 1, \ldots, k \\ \text{and } cl \in unf_{\{\phi(c')\},\Omega(\phi(Q^\star)),\Delta_{\phi(Q^\star)}}\{c_1, \ldots, c_k\}. \end{array}$$

Then, being $c_i = \phi(c'_i)$ for $i = 1, \ldots, k$, $cl \in unf_{\{\phi(c')\},\Omega(\phi(Q^\star)),\Delta_{\phi(Q^\star)}}\{\phi(c'_1), \ldots, \phi(c'_k)\}$ and this together with (1) implies that $cl \in \phi(unf_{\{c'\},\Omega(Q),\Delta_Q}\{c'_1, \ldots, c'_k\})$. Furthermore, since $\phi(Id_{\Omega(\phi(Q^\star))}) = Id_{\Omega(\phi(Q^\star))}$ and by the inductive hypothesis, $T^{cs}_{\phi(Q^\star)} \uparrow n-1 \subseteq \phi(T^{cs}_Q \uparrow n-1 \setminus S)$, it follows that for $i = 1, \ldots, k$, $c'_i \in (T^{cs}_Q \uparrow n-1 \setminus S) \cup Id_{\Omega(\phi(Q^\star))}$. Hence, by monotonicity of unf:

$$cl \in \phi(unf_{\{c'\},\Omega(Q),\Delta_Q}\ (T^{cs}_Q \uparrow n-1 \cup Id_{\Omega(\phi(Q^\star))})) \subseteq \phi(T^{cs}_Q \uparrow n).$$

To conclude the proof that $cl \in \phi(T^{cs}_Q \uparrow n \setminus S)$ we only need to show that $cl \notin \phi(S)$. We do this by considering the three cases in the definition of the set S.

- *(i) For any $i = 1, \ldots, k$, $c'_i \in (T^{cs}_Q \uparrow n-1 \setminus S) \cup Id_{\Omega(\phi(Q^\star))}$, then $s_i \cap \kappa(P) = \emptyset$. Moreover if $c'_i \in T^{cs}_Q \uparrow n-1 \setminus S$ then $Pred(B'_i) \notin \Delta_Q \cap \kappa(P)$. Therefore by definition of the unfolding rule, $s \cap \kappa(P) = \emptyset$.*

- *(ii) Since $c' \in Q^\star$, by definition 2.4 $Pred(A) \notin \Delta_Q \cap \kappa(P)$.*

- *(iii) By definition of S, if $c'_i \in (T^{cs}_Q \uparrow n-1 \setminus S)$ then $Pred(\tilde{L}_i) \cap \Sigma_Q = \emptyset$. By definition of $\phi(Q^\star)$, $\Omega(\phi(Q^\star)) \cap \Sigma_Q = \emptyset$ and hence, if $c'_i \in Id_{\Omega(\phi(Q^\star))}$ then $Pred(\tilde{L}_i) \cap \Sigma_Q = \emptyset$. By these equalities it follows that $Pred(\tilde{L}_1, \ldots, \tilde{L}_k) \cap \Sigma_Q = \emptyset$.*

$\left(T^{cs}_{\phi(Q^\star)} \uparrow n \supseteq \phi(T^{cs}_Q \uparrow n \setminus S)\right)$.
Assume that

$$\phi(cl) = \phi((A \text{:-} s \ \Box \ \tilde{L}_1, \ldots, \tilde{L}_k)\vartheta) \in \phi(T^{cs}_Q \uparrow n \setminus S).$$

By definition of $T^{cs}_Q \uparrow n$, *there exist*

$$c = A\,\text{:-}\,B_1, \ldots, B_k \in Q,$$
$$c_i = B'_i\,\text{:-}\,s_i \,\square\, \tilde{L}_i \in T^{cs}_Q \uparrow n-1 \cup Id_{\Omega(Q)},\ i = 1, \ldots, k$$

such that $cl \in unf_{\{c\},\Omega(Q),\Delta_Q}\{c_1, \ldots, c_k\}$. *Let us assume without loss of generality that there exists* $r \leq k$ *such that* $c_1, \ldots, c_r \in T^{cs}_Q \uparrow n-1 \setminus Id_{\Omega(Q)}$ *and* $c_{r+1}, \ldots, c_k \in Id_{\Omega(Q)}$. *Since* $cl \notin S$, *by the same arguments used above, it can be shown that*

(i) $Pred(A) \notin \Delta_Q \cap \kappa(P)$;

(ii) For $i = 1, \ldots, r$, $Pred(B_i) \notin \Delta_Q \cap \kappa(P)$ *and* $s_i \cap \kappa(P) = \emptyset$;

(iii) For $i = 1, \ldots, k$, $Pred(\tilde{L}_i) \cap \Sigma_Q = \emptyset$.

Now, for $i = 1, \ldots, r$, *from (ii) it follows that* $c_i \notin S$ *and hence, by the inductive hypothesis,* $\phi(c_i) \in T^{cs}_{\phi(Q^\star)} \uparrow n-1$. *For* $j = r+1, \ldots, k$, *we can show that* $\phi(c_j) \in Id_{\Omega(\phi(Q^\star))}$. *Recall that* $c_j \in Id_{\Omega(Q)}$ *and observe that, by definition of* $\phi(Q^\star)$, *if* $p \in \pi(Q^\star) \cap \Omega(Q)$ *and* $\phi(p) \notin \Omega(\phi(Q^\star))$ *then* $p \in \Sigma_Q$. *But then, since by (iii),* $Pred(B_j) = Pred(\tilde{L}_j) \notin \Sigma_Q$, *it follows that* $\phi(c_j) \in Id_{\Omega(\phi(Q^\star))}$. *Now, from (i), we have that* $\phi(c) \in \phi(Q^\star)$. *Then, being* $cl \in unf_{\{c\},\Omega(Q),\Delta_Q}\{c_1, \ldots, c_k\}$, *by (1) and injectivity of* ϕ, $\phi(cl) \in unf_{\{\phi(c)\},\Omega(\phi(Q^\star)),\Delta_{\phi(Q^\star)}}\{\phi(c_1), \ldots, \phi(c_k)\}$. *Hence, we can conclude being*

$$\begin{array}{rcl} \phi(cl) \in unf_{\{\phi(c)\},\Omega(\phi(Q^\star)),\Delta_{\phi(Q^\star)}}\{\phi(c_1), \ldots, \phi(c_k)\} & \subseteq & T^{cs}_{\phi(Q^\star)}(T^{cs}_{\phi(Q^\star)} \uparrow n-1) \\ & = & T^{cs}_{\phi(Q^\star)} \uparrow n \end{array}$$

■

Lemma 3.17 Let $\langle \Sigma_P, \Delta_P, \Theta_P \rangle$-$P$,$\langle \Sigma_Q, \Delta_Q, \Theta_Q \rangle$-$Q$ be differential programs, $P \lhd Q$ be the differential program $\langle \Sigma, \Delta, \Theta \rangle$-$(\xi(P) \cup \phi(Q^\star))$ where $Q^\star$, ξ, ϕ is defined according to definition 2.4. Moreover let R and S be defined as in lemma A.8 and $[\![Q]\!]^r$ be defined according to definition 3.15. Then

1. $[\![\xi(P)]\!] = [\![P]\!] \setminus R$ where $R = \{H\,\text{:-}\,s \,\square\, \tilde{B} \in \mathcal{C}_\Delta \mid Pred(\tilde{B}) \cap \Sigma_P \not\subseteq \kappa(Q)\}$
2. $[\![\phi(Q^\star)]\!] = [\![Q]\!]^r$

Proof. Again we prove 1. and 2. separately.

Proof of 1. Observe that if the predicate symbol p is defined in P, then p is not renamed by ξ and hence

$$\kappa(P) \cap \iota(P) \quad = \quad \kappa(\xi(P)) \cap \iota(\xi(P)). \tag{1}$$

Given a set Ψ of predicate symbols recall that
$\mathcal{D}(\Psi) = \{H\,\text{:-}\,s \,\square\, \tilde{B} \in \mathcal{C}_\Delta \mid Pred(H) \in \Psi\}$.

Moreover observe that, by definition of α (definition 3.8), given a set of cs-clauses A and for R defined as before we have

$$\alpha(A) \setminus R \quad = \quad \alpha(A \setminus R) \tag{2}$$

We have the following equalities

$$\begin{array}{lcl}
[\![\xi(P)]\!] & = & \text{by definition of } [\![\cdots]\!] \\
\alpha(T^{cs}_{\xi(P)} \uparrow \omega) \setminus \mathcal{D}(\iota(\xi(P))) & = & \text{by definition of } \uparrow \omega \\
\alpha(\cup_{n\geq 0} T^{cs}_{\xi(P)} \uparrow n) \setminus \mathcal{D}(\iota(\xi(P))) & = & \text{by lemma A.8} \\
\alpha\left(\cup_{n\geq 0}(T^{cs}_P \uparrow n \setminus R)\right) \setminus \mathcal{D}(\iota(\xi(P))) & = & \text{by definition of } \uparrow \omega \text{ and by (1)} \\
\alpha(T^{cs}_P \uparrow \omega \setminus R) \setminus \mathcal{D}(\iota(P)) & = & \text{by 2 and by set theory} \\
(\alpha(T^{cs}_P \uparrow \omega) \setminus \mathcal{D}(\iota(P))) \setminus R & = & \text{by definition of } [\![\cdot]\!] \\
[\![P]\!] \setminus R &&
\end{array}$$

and the thesis holds.

Proof od 2. Note that a predicate symbol p is defined in $Q^\star$ iff p is defined in Q and $p \notin \Delta_Q \cap \kappa(P)$. Then by definition of ϕ we have

$$\phi(p) \in \kappa(\phi(Q^\star)) \wedge \phi(p) \in \iota(\phi(Q^\star)) \quad \text{ iff } p \in \kappa(Q^\star) \wedge p \in \iota(Q) \cup (\Sigma_Q \cap \kappa(P))$$

and, for $\mathcal{D}$ defined as before,

$$\mathcal{D}(\iota(\phi(Q^\star))) = \phi\left[\mathcal{D}(\iota(Q)) \cup \mathcal{D}(\Sigma_Q \cap \kappa(P))\right]. \tag{1}$$

Let S be defined as in lemma A.8, i.e. $S = \{H \text{:-} s \,\square\, \tilde{B} \in \mathcal{C}_\Delta \mid s \cap \kappa(P) \neq \emptyset$ or $Pred(H) \in \Delta_Q \cap \kappa(P)$ or $Pred(\tilde{B}) \cap \Sigma_Q \neq \emptyset\}$. We first show that, given a set of cs-clauses A, for such an S we have

$$\alpha(A \setminus S) = \alpha(A) \setminus S \tag{2}$$

The equality is clear for the clauses which are in S because either the second or the third condition in the definition of the set S holds. Then we will consider S as defined only by the first condition (i.e. $s \cap \kappa(P) \neq \emptyset$). By definition 3.8, $cl = H : -s\Box\tilde{B} \in \alpha(A \setminus S)$ iff $cl \in A$, $s \cap \kappa(P) = \emptyset$ and $\nexists\ cl' = H : -s'\Box\tilde{B} \in A \setminus S$ with $s' \subset s$ (we consider the same $H : -\tilde{B}$ instead than an $\approx$-equivalent one for the sake of simplicity). Note that, since $s' \cap \kappa(P) = \emptyset$, $cl' \in A \setminus S$ iff $cl' \in A$. Therefore $cl \in \alpha(A \setminus S)$ iff $cl \in \alpha(A) \setminus S$ and the thesis holds. We have then the equalities

$$
\begin{array}{lcl}
[\![\phi(Q^\star)]\!] & = & \text{by definition of } [\![\cdot]\!] \\
\alpha(T^{cs}_{\phi(Q^\star)} \uparrow \omega) \setminus \mathcal{D}(\iota(\phi(Q^\star))) & = & \text{by definition of } \uparrow \omega \\
 & & \text{and by lemma A.8} \\
\alpha\left(\phi(T^{cs}_Q \uparrow \omega \setminus S)\right) \setminus \mathcal{D}(\iota(\phi(Q^\star))) & = & \text{by (1)} \\
\alpha\left(\phi(T^{cs}_Q \uparrow \omega \setminus S)\right) \setminus \phi(\mathcal{D}(\iota(Q)) \cup \mathcal{D}(\Sigma_Q \cap \kappa(P))) & = & \text{by (2) and by def. of } \phi \\
\left(\alpha(\phi(T^{cs}_Q \uparrow \omega)) \setminus \phi(S)\right) \setminus \phi(\mathcal{D}(\iota(Q)) \cup \mathcal{D}(\Sigma_Q \cap \kappa(P))) & = & \text{by definition of } \phi \\
\left(\phi(\alpha(T^{cs}_Q \uparrow \omega)) \setminus \phi(S)\right) \setminus \phi(\mathcal{D}(\iota(Q)) \cup \mathcal{D}(\Sigma_Q \cap \kappa(P))) & = & \text{by set theory and def. of } \phi \\
\phi\left(\alpha(T^{cs}_Q \uparrow \omega) \setminus \mathcal{D}(\iota(Q))\right) \setminus \phi\left(S \cup \mathcal{D}(\Sigma_Q \cap \kappa(P))\right) & = & \text{by definition of } [\![\cdot]\!] \\
\phi\left([\![Q]\!] \setminus (S \cup \mathcal{D}(\Sigma_Q \cap \kappa(P)))\right) & = & \text{by definition of } S, \mathcal{D} \text{ and } {}^r \\
\phi([\![Q]\!]^r) & = & \text{by the following observation} \\
[\![Q]\!]^r & &
\end{array}
$$

By definition of $[\![Q]\!]^r$, if $p \in Pred([\![Q]\!]^r)$ then $p \notin \iota(Q) \cup (\kappa(P) \cap \Sigma_Q) \cup (\Sigma_Q \cap \kappa(Q))$ and hence, by definition of ϕ, $\phi(p) = p$ holds. Thus $\phi([\![Q]\!]^r) = [\![Q]\!]^r$ and this completes the proof. ■