Differential Logic Programs:
Programming Methodologies and Semantics

A. Bossi   M. Bugliesi

Dipartimento di Matematica Pura ed Applicata

Università di Padova via Belzoni 7, 35131 Padova, Italy

{isa@zenone,michele@goedel}.math.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

We introduce the notion of *differential logic programs* and we define an operator for composing them in a hierarchical fashion. The semantics of this composition operator is reminiscent the semantics of inheritance of the object oriented paradigm. Similarly to classes in that paradigm, differential programs can be organized in *isa* schemas where each component inherits or redefines, modifying them, the predicates defined in the components that are placed higher up in the schema. We demonstrate the use of this form of composition as a programming methodology that enhances reusability, code sharing and information hiding.

We define a proof theory and a model theory for the composition of differential programs and we prove that the two theories coincide. We also define a compositional and fully abstract semantics for differential programs and we address the importance of this semantics as a formal tool for reasoning on the computational properties of differential programs and their composition.

A preliminary version of this paper appeared in [5].

1 Introduction

Program composition has been proposed as a tool for modular logic programming by many authors in the literature. It was first introduced by R. O'Keefe in [26]: his goal was to provide a formal account of the software engineering approach to programming that seeks to develop programs incrementally by defining several components, together with their interfaces, and then by composing the components. He formalized this approach by interpreting logic programs as elements of an algebra and by modeling their composition in term of the operators of the algebra. Following an orthogonal direction, in [22] Miller proposed a modular logic programming language that uses embedded (intuitionistic) implications as operators for building and composing modules in a purely logical setting. His idea was to enrich logic programming with more adequate *linguistic* mechanisms for programming-in-the-small and then to tailor those mechanisms to attack the problems of programming-in-the-large.

For programming-in-the-large, viewing modular programming as program composition offers several advantages over embedded implications. Firstly, being inherently a *meta-linguistic* mechanism, it provides a powerful tool for structuring programs that does not require any extension of the theory of Horn clauses. It is also highly flexible as new composition mechanisms can be accounted for by simply introducing a corresponding operator in the algebra or by combining the existing ones. Finally, when coupled with mechanisms for specifying the interfaces between components, it allows one to model powerful forms of encapsulation and information hiding. Several proposals in the literature follow this approach to modular logic programming. The "open programs" introduced by Bossi et al. in [7], the modular frameworks of Mancarella and Pedreschi [19], Gaifman and Shapiro [13], and of Brogi et al. [9] can in fact be seen as different formulations of this idea.

Our paper contributes to this line of research. Building on the idea of constructing programs by composition, we consider a novel and general composition operator that is amenable to several specializations. The semantics of this composition operator is reminiscent the semantics of inheritance of the object oriented paradigm. Similarly to classes in that paradigm, differential programs can be organized in *isa* schemas where each component inherits or redefines – modifying them – the predicates defined in the components that are placed higher up in the schema.

The aim of the paper is twofold. We illustrate the use of this form of program composition as a powerful abstraction tool for modular programming. We show that this programming practice is very effective as it supports very naturally a structured approach to software development and it provides adequate tools for code sharing and code reuse among the components of a hierarchy. We also study the semantic properties of our composition operator. In that direction, we first define a proof theory and a model theory for the composition of differential programs, and we show that the two theories coincide. Then, we define a compositional and fully abstract semantics for differential programs. Being compositional, this semantics allows us to isolate, within a hierarchy, an abstract characterization of the computational behavior of every component program. Being fully abstract, it guarantees that this characterization contains the least amount of information needed to make it correct. Owing to these properties, the fully abstract semantics represents an effective tool for reasoning on the computational properties of differential programs and their composition. In particular, since it provides an exact characterization of the relation of computational equivalence between two programs, it can be used to give a formal justification for some of the basic operations that should be accounted for by

every paradigm of programming-in-the-large: reusability - equivalent components reused within different hierarchies – transformation and optimization – replacement of equivalent components – separate compilation, . . . etc.

We organize the rest of our paper as follows. In Section 2 we present the salient features of our composition operator and we discuss its informal semantics by means of simple examples. In Section 3 we introduce the proof theory as well as the model theory and we show their equivalence. In Section 4 we define the compositional semantics and we prove it fully abstract. We conclude in Section 5 putting our proposal in perspective and discussing related research. A final appendix contains proofs of some technical results needed in the rest of the paper.

2 Differential Logic Programs by Example

In this section we present an informal account of the programming technique that arises from the use of differential logic programs. One of the central ideas of this approach to programming is that writing a program is an incremental process of *differentiation* that builds new program components by extending or modifying the behavior of existing components.

The following example illustrates an application of this methodology in the development of a window editor like EMACS.

Emax: an Emacs-like Editor

We start by assuming that the host system supports the basic functionalities needed in the design of a display editor: file system operations such as open, close, save, . . . etc. and buffer management facilities such as cursor moves. Assuming Prolog as the logic language at disposal, we first define a generic interface module `EDITOR` that provides the code needed 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
                ...................
```

Given this generic interface module, `EMAX` – the display editor – will consist of 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 these 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 these events raise exceptions that are forwarded to `EMAX` as queries of the form `exception(Buffer, Cause)`.

---

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

```
EMAX %% is_a EDITOR
     %% inherits open/2, quit/0, save/2

%%%%  Keystroke Events
   ks_event(Buff, ['C-x'|X]) :- get_fname(Buff, File),
                                  cx_action(Buff, File, X).
   ks_event(Buff, [C]) :- echo(C, Buff),
                            match(Buff, C).
%%%%  Handlers
   cx_action(Buff, File, 'C-f') :-  open(File, Buff).
   cx_action(Buff, File, 'C-c') :-  exit(Buff, File).
   exit(Buff, File) :- query_user('save changes ?', Ans),
                         Ans = 'yes',
                         save(Buff, File), quit.
   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#), quit.
```

The behavior of the handler `cx_action` for `C-x` events should be obvious from the definition: the sequence `C-x,C-f` opens a new file whereas `C-x,C-c` exits the current editing session. `Match` 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`.

Note that `EMAX` contains no definition for `open`, `save`, `quit`, although these primitives are invoked by the event handlers. This is because the intention is to use `EMAX` and `EDITOR` as two components of the hierarchy `EMAX isa EDITOR`, where `EMAX` inherits the definition of the primitives from its (immediate) ancestor `EDITOR`. Further modules, composed onto this hierarchy, will in their turn inherit (possibly modifying them) the definitions of the existing components.

Consider for instance defining a new editor that handles control sequences other than the `C-x` sequence handled by `EMAX`. The way the new editor would be implemented is by extending the hierarchy `EMAX isa EDITOR` with a new module that introduces the new clauses for `ks_event` and inherits (monotonically) the remaining clauses as well as the other predicates defined by `EMAX` and `EDITOR`. In the extended hierarchy, the clauses defining `ks_event` would be thus distributed among the components and shared among them. The following module, implementing a LaTeX-mode for `EMAX`, illustrates this situation.

```
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 $').
```

A different way that `EMAX` may be specialized is by associating different responses with the
`C-x` class of events, or else by modifying the behavior of the handler for certain events. The
following specialization of `EMAX` motivates and illustrates this scenario.

```
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 `EMAX` with the revision control facilities of 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. Consider composing `RCS-EMAX` onto the hierarchy
`EMAX isa EDITOR`: clearly the intention is that `exit` and `save` behave deterministically in the
new hierarchy and, consequently, the two definitions contained in the new module should both
override, rather than extend, the corresponding definitions in `EMAX` and `EDITOR`.

Also, there are reasons for distinguishing between the behavior of the two predicates. Con-
sider first the case of a `C-x,C-c` event forwarded to `RCS-EMAX`: the event activates the handler
`cx_action` in `EMAX`, but now one would expect that `cx_action` invokes the `exit` and `save`
routines of `RCS-EMAX`. This behavior is well-known in object oriented systems and results from
the combination of overriding and dynamic binding: here `RCS-EMAX` plays the role of *self*, being
the receiver of the request, and what we need is to have late binding for the call to `exit` in
the handler `cx_action`. For `save`, instead, we would like to enforce static binding to guarantee
that the call to this predicate in `RCS-EMAX` be bound to the local definition.

Static binding for `save` ensures also a safe treatment of exceptions. Consider for instance
the case of a crash exception forwarded to `RCS-EMAX`. The event, in this case, activates the
the definition of `exception` in `EMAX`. Now there are good reasons for letting `EMAX` and `EDITOR`
handle the exception as, upon a system crash, it is often useless to try and do anything else
than dumping a copy of the file as it is. But this behavior is automatic if we enforce static
binding for `save`: the call to the predicate in `EMAX` will, in this case, be bound statically to the
definition that `EMAX` inherits from `EDITOR`.

3   Hierarchies of Differential Logic Programs

In this section, we introduce the notions differential program and hierarchy of programs, define
the rules of query evaluation in a hierarchy, and study other computational aspects relative
to the *isa* composition. Throughout this and the following section, we will assume that the
hierarchies that are formed with differential programs are *linear*. This assumption will be lifted

---

[2]RCS is the Revision Control System developed by W. F. Tichy (see [28]).

5

in Section 5 where we show that that it does not involve any loss of generality as the cases of tree-like and DAG-like hierarchies can be characterized semantically, reasoning on linear hierarchies.

We start by introducing the definition of differential programs, programs that are built over an alphabet of annotated predicate symbols.

Definition 3.1 (Differential program) Let a first order alphabet be defined as a triple $\langle V, F, \Pi \rangle$ of denumerable sets of, respectively, variable, function and predicate symbols. An *annotated alphabet* A is a first order alphabet where the set Π is partitioned into four sets STAT, DYN, EXT and INT called, respectively, the *statically inheritable, dynamically inheritable, extensible* and *internal* predicates of A.
A definite logic program P over an alphabet A is a *differential program* if and only if A is annotated and all of the internal predicates of A that occur in P are defined[3] in P.

To ease the notation, we fix ahead an annotated alphabet A and consider differential programs over this alphabet avoiding to mention A altogether. Similarly, we refer to the predicate symbols from STAT, DYN, EXT and INT with the understanding that these symbols are taken from the same annotated alphabet.

We denote with *Preds(P)*, the set of predicate symbols occurring in P and with STAT(P), DYN(P) and EXT(P), respectively, the *statically inheritable* (static), *dynamically inheritable* (dynamic), and *extensible* predicates of the program. These three sets comprise all of the *public* predicates of the program, that the program shares with the other components of the hierarchies it is part of. We will henceforth write $p \in Pub(P)$ to state that p is one of the public predicates of P. The remaining predicates of P, denoted with INT(P), are the *internal* predicates, that the program defines for its own private usage. Being private to the program, we assume that P defines these predicates.

The structure of an *isa* hierarchy is defined by the following productions:

$$H ::= P \mid P \ isa \ H$$

where P and H are meta-linguistic variables standing respectively for a differential program and a hierarchy of differential programs over the same annotated alphabet. Given the *isa* hierarchy $H = P_n \ isa \ (\cdots \ isa \ (P_2 \ isa \ P_1) \cdots)$, we say that P_j is an ancestor of P_i (dually, P_i is a heir of P_j) whenever $j < i$ and we call P_n and P_1 respectively the *leaf* and *root* elements of H.

The formation of an *isa* hierarchy is subject to certain "well-formedness" constraints that we first state, in the next definition, and then motivate.

Definition 3.2 (Well-formed Hierarchy)
A differential program P is a well-formed *isa* hierarchy provided that all of the predicates symbols in STAT(P) are defined in P.

If H is a well-formed *isa* hierarchy and P a differential program, then $P \ isa \ H$ is a well-formed hierarchy if and only if:

1. every predicate in STAT(P) is defined either in P or in one of the components of H,

---

[3]We say that a predicate p is defined in a program P if and only if P contains at least one clause whose head's predicate symbol is p.

2. $\text{INT}(P) \cap \text{INT}(H) = \emptyset$.

We have denoted with $\text{INT}(H)$ the set of internal predicates of H, i.e. the union of the sets of internal predicates in the components of H. With the same understanding we will extend all the notation and definitions we have introduced for differential programs to hierarchies of differential programs.

The first well-formedness constraint guarantees that it be possible to resolve the calls to every static predicate using static binding. The second condition, in its turn, requires that in every well-formed hierarchy the set of internal predicates of the components be pairwise disjoint. This is a fairly mild assumption that involves no loss of generality. In fact, since the reference to the internal predicates of a program are resolved locally to that program, the choice of the names of these predicates is immaterial to the behavior of the program. This property of internal predicates justifies also the introduction of a notion of standardization apart between differential programs: we say that two differential programs P and Q are INT-standardized apart if and only if they share no common internal predicates ($\text{INT}(P) \cap \text{INT}(Q) = \emptyset$).

We will henceforth consider only well-formed hierarchies and, within these hierarchies, we will consider only the evaluation of goals whose predicate symbols are public for (hence, exported by) at least one of the component programs.

3.1 Proof Theory

The behavior of a call to a predicate in a hierarchy depends on the annotation of that predicate. *Dynamic* and *static* predicates subject to overriding and overriding, in its turn, is based on the existence of an overriding definition: each component of the hierarchy inherits a predicate from the closest ancestor in the hierarchy that contains a definition for that predicate. More precisely, given the hierarchy

$$P_n \; isa \; \ldots \; isa \; P_2 \; isa \; P_1$$

for every i, P_i inherits a predicate from an ancestor P_j, if and only if neither P_i nor any intervening component P_k $(j < k < i)$ in the hierarchy defines that predicate. Counterwise, extensible predicates are inherited monotonically: each component inherits a predicate from *all of* its ancestors in the hierarchy. Furthermore, dynamic and extensible predicates are evaluated with dynamic binding, whereas the references to a static predicate are always resolved using static binding.

   The combination of these mechanisms results into the following rule for evaluating a call. Given the *isa* hierarchy $P_n \; isa \; \ldots \; isa \; P_2 \; isa \; P_1$, consider evaluating an atomic goal G in the component P_j and assume that p is the predicate symbol of G. The evaluation of G proceeds according to the criteria outlined below:

1. if p is an internal predicate, select a clause from the (local) definition contained in P_j;

2. if p is a static predicate, select a clause from the closest ancestor of P_j that defines p;

3. if p is a dynamic predicate, select a clause from the closest ancestor of P_n that defines p (independently of the component P_j where the call occurs);

4. if p is an extensible predicate, select a clause from any of the components that defines p.

These rules are formalized below in Natural Deduction style following the same idea used in [21]. We denote with bold upper-case letters, like $\mathbf{G}$ and $\mathbf{B}$, conjunctions of atoms, with $\mathbf{x}, \mathbf{t}$

tuples of, respectively, variables and terms. Where H denotes a hierarchy, the notation $H \vdash \mathbf{G}$ should be read "$\mathbf{G}$ succeeds in H". Evaluating an initial goal in H triggers the evaluation of other subgoals that are evaluated from within a specific component of the hierarchy. The notation $j, H \vdash_\vartheta \mathbf{G}$ should be read "$\mathbf{G}$ succeeds in H with substitution ϑ when $\mathbf{G}$ is invoked from the j-th component of H".

The evaluation of a goal $\mathbf{G}$ in a hierarchy starts from the leaf of the hierarchy, provided that all the predicates symbols occurring in $\mathbf{G}$ belong to the set of public predicates of the hierarchy. Letting $Pred(\mathbf{G})$ denote the set of predicate symbols of $\mathbf{G}$, the rule is as follows:

$$\frac{n, \ P_n \ isa \cdots isa \ P_1 \vdash_\vartheta \mathbf{G}}{P_n \ isa \cdots isa \ P_1 \vdash \mathbf{G}}$$

provided that $Pred(\mathbf{G}) \subseteq Pub(P_n \ isa \cdots isa \ P_1)$.

The rule for conjunctive goals splits the evaluation on each of the conjuncts. For $\mathbf{G} = \mathbf{G}_1, \mathbf{G}_2$:

$$\frac{j, \ P_n \ isa \cdots isa \ P_1 \ \vdash_\theta \mathbf{G}_1 \qquad j, \ P_n \ isa \cdots isa \ P_1 \ \vdash_\sigma \mathbf{G}_2 \theta}{j, \ P_n \ isa \cdots isa \ P_1 \ \vdash_{\theta\sigma} \mathbf{G}_1, \mathbf{G}_2}$$

Note that, since the order of the atoms in a conjunction is irrelevant, the choice of the split $\mathbf{G}_1, \mathbf{G}_2$ of $\mathbf{G}$ is free and hence the proof of conjunctive goals is independent of any selection rule.

If $\mathbf{G}$ is an atomic goal, say $p(\mathbf{t})$, then the annotation for the predicate p determines the different ways of selecting a clause in the components of the hierarchy. The proof rule is as follows.

$$\frac{k, \ P_n \ isa \cdots isa \ P_1 \vdash_\sigma \mathbf{G}\vartheta}{j, P_n \ isa \cdots isa \ P_1 \ \vdash_{\vartheta\sigma} p(\mathbf{t})}$$

provided that $k \in \{1, .., n\}$ is a component-index in H, $p(\mathbf{t}') \leftarrow \mathbf{G}$ is a clause in the component P_k, $\vartheta = mgu(p(\mathbf{t}), p(\mathbf{t}'))$, and one of the following conditions holds:

$$
\begin{aligned}
&(1) \ p \in \text{STAT} \quad \wedge \quad k = max \ \{i \leq j \mid P_i \text{ defines } p\} \\
&(2) \ p \in \text{DYN} \quad \wedge \quad k = max \ \{i \leq n \mid P_i \text{ defines } p\} \\
&(3) \ p \in \text{EXT} \quad \wedge \quad k \in \{i \mid P_i \text{ defines } p\} \\
&(4) \ p \in \text{INT} \quad \wedge \quad k = j
\end{aligned}
$$

As it is customary in logic programming, the evaluation of a goal succeeds when a finite number of applications of the above rules leads to the evaluation of an empty goal. This is formalized by introducing the following rule:

$$\frac{}{j, \ P_n \ isa \cdots isa \ P_1 \vdash_\varepsilon \square}$$

where $\square$ denotes the empty goal and ε the empty substitution.

The operational semantics for the hierarchical composition of differential logic programs can now be defined as follows. Call *isa*-proof for $H \vdash G$ a proof-tree T rooted at $H \vdash G$ such that:

1. every leaf node of T is labeled by $j, H \vdash_\varepsilon \square$, where j in an index of the components of H;

2. every internal node is labeled by one of the upper sequents of the proof rule whose lower sequent is the label of the parent of the node;

8

3. the *mgu*'s computed at each step of the proof are idempotent and the clause selected at each step is standardized apart with respect to all the variables that occur in (all of the branches of) the tree at the step the selection is made.

Remarks. Note that, strictly speaking, we have introduced two relations of provability, "$\vdash$" and "$\vdash_\vartheta$": the former is defined in terms of the latter and it abstracts over it hiding the substitution computed during the proof. This choice is intentional as it allows us to define the relation of provability in a hierarchy of differential programs independently of any notion of substitution: substitutions are needed to carry out the proof, but we don't want to observe them at the end of the proof. As in logic programming, we are thus defining the (non ground) *success set* of a hierarchy (or a program, for that matter) as the observables of a computation[4].

3.2   Model Theory

To define a corresponding model theory for the composition of differential programs, we introduce a mapping from *isa* hierarchies to corresponding (and equivalent) logic programs whose proof theory can be stated in terms of SLD-resolution. In this section we introduce this mapping and then study its formal properties in full details.

For the remaining of this and the following sections, we assume familiarity with the standard notions of logic programming as introduced in [3] and [18]. The notation $\mathbf{G} \overset{\vartheta}{\rightsquigarrow}_{P,R} \mathbf{B}$ stands for a (partial) SLD derivation in the program P from $\mathbf{G}$ to the resolvent $\mathbf{B}$, where R is the selection rule and ϑ is the composition of the mgu's used in the derivation. Similarly, we write $\mathbf{G} \overset{\vartheta}{\rightsquigarrow}_P \square$ to denote an SLD refutation for $\mathbf{G}$ leading to the empty resolvent $\square$: R is omitted in this case because ϑ is independent (up to renaming) of R when the derivation is a refutation.

The mapping from *isa* hierarchies of differential programs to logic programs is defined inductively on the structure of the hierarchies, in terms of a syntactic composition operator on differential programs. Given a well-formed hierarchy P *isa* H, the mapping associates it with the differential program obtained by composing P with the differential program associated (through the mapping itself) with the hierarchy H.

We next introduce the operator of syntactic composition. Composing two programs is the result of performing three combined operations on the clauses of the components: union, subsetting and renaming. Union and renaming realize the effect of inheritance in the corresponding *isa* hierarchy of the two programs; the subsetting, in its turn, mimics the functionalities of overriding. The union $\cup$ of two INT-standardized apart differential programs is defined as the program obtained by taking the union of the two set of component clauses. The remaining operations needed in the composition are defined below.

Given a differential program P, let $Defn(P)$ denote the set of predicates defined in P.

Definition 3.3 (◁-composition) Let P and Q be two differential programs such that P *isa* Q is a well-formed hierarchy. Let $\phi(Q^\star)$ denote the new differential program obtained as follows:

- $Q^\star = \{A \leftarrow G \in Q \mid Pred(A) \nsubseteq \text{DYN}(Q) \cap Defn(P)\}$;

- ϕ renames the predicates in $\text{STAT}(Q) \cap Defn(P)$ with *new* names from the set INT of internal predicates which do not occur either in P or Q.

---

[4]See [6] for a full treatment of computed answer substitutions as observables.

Then $P \lhd Q$ is the differential program $P \cup \phi(Q^\star)$.

The subsetting $\star$ on Q removes all the clauses of Q defining dynamic predicates that are also defined in P. As we said, this removal mimics the effect of overriding the dynamic predicates of Q in the hierarchy P isa Q leaving in $Q^\star$ only the definitions that are not overridden by the corresponding definitions in P.

The use of the renaming ϕ is dictated by the need to guarantee that the calls to the static predicates of $Q^\star$ be bound to the definitions that $Q^\star$ provides for these predicates, thus avoiding possible clashes with the names (and definitions) of the static predicates of P. This way, the renaming realizes the effect of the static binding that applies to the static predicates of the hierarchy P isa Q. To see that, let p a renamed static predicate of Q. Since it is renamed, p must be defined in P: hence, by effect of the renaming, each call to p coming from the clauses of Q that are left in $Q^\star$ is renamed and bound to the (correspondingly renamed) definition of $\phi(Q^\star)$. Counterwise, the definition of p that is "exported" by $P \lhd Q$ is the definition contained in P (which is not renamed).

Finally, note that every static predicate of Q that is renamed by ϕ must be defined in P and that all the *new* internal predicates (those deriving from the renamings of static predicates) are defined in $P \lhd Q$. Hence, $P \lhd Q$, seen as a one component hierarchy, is well-formed. Now we define the mapping between hierarchies of differential programs and differential programs as follows.

Definition 3.4 (Mapping $\sharp$) For every well-formed hierarchy, let $\sharp$ be the mapping defined as follows:

- $P^\sharp = P$ for every differential program P;

- $(P\ isa\ H)^\sharp = P \lhd \rho(H^\sharp)$ where ρ is a renaming of the internal predicates of $H^\sharp$ such that $\rho(H^\sharp)$ and P are INT-standardized apart.

It is easy to see that if P isa H is well-formed, then so is P isa $\rho(H^\sharp)$. Hence the mapping $\sharp$ is well defined. The following example illustrates the syntactic counterpart of the hierarchy of the components of the editor discussed in section 2.

Example 3.5 The following program is the result of the transformation $\sharp$ applied to the hierarchy `RCS-EMAX isa EMAX isa EDITOR`.

```
%%%% (RCS-EMAX isa EMAX isa EDITOR)♯

  open(File, Buff) :- .....                         %from EDITOR
  move(Buff, 'up') :- .....                         %from EDITOR
  save@EDITOR(Buff,File)  :- .....                  %from EDITOR renamed

  exception(Buff, crash) :- save@EDITOR(Buff, *Buff*),  %from EMAX renamed
                            quit.

  ks_event(Buff, ['C-x'|X]) :- get_fname(Buff, File),   %from EMAX
                               cx_action(Buff, File, X).
  ks_event(Buff, [C])       :- echo(C, Buff),           %from EMAX
                               match(Buff, C).

  cx_action(Buff, File, 'C-c') :-  open(File, Buff).    %from EMAX
```

```
cx_action(Buff, File, 'C-f') :- exit(Buff, File).     %from EMAX

match(Buff, '}') :- find_matching('{', Buff, Pos),     %from EMAX
                    highlight(Pos, Buff).
match(_, '}')    :- warning('mismatched brace').       %from EMAX
match(_, C).                                            %from EMAX

exit(Buff, File) :-  get_version_num(File,Vn),         %from RCS-EMAX
                     save(Buff, File:Vn), quit.
save(Buff, File:N) :- diff(Buff, File:N, DL),          %from RCS-EMAX
                      N1 is N+1,
                      save_changes(DL, File:N1).
```

The occurrences of the static predicate `save` in the clauses coming from `EDITOR` and `EMAX` have been renamed, with the internal predicate `save@EDITOR` to save clashes with the definition contained in `RCS-EMAX`. The definition of the dynamic predicate `exit` in `EMAX` has been removed because overridden by the corresponding definition contained in `RCS-EMAX`.

We are now ready to define the declarative semantics of an *isa* hierarchy of differential programs. For every differential program P denote with M_P the least Herbrand model of P, and with $M(P)$ the projection of M_P on the set of public predicate symbols of P. Formally,

$$M(P) \quad = \quad \{A \in M_P \mid Pred(A) \subseteq Pub(P)\}.$$

The set $M(P)$ represents the natural counterpart of the least Herbrand model of a logic program: owing to the on the public predicates, the "least model" of a differential program does not depend on the names of the internal predicates of the program. The "least model" of a hierarchy is defined according to the same idea as suggested by the following definition.

Definition 3.6 (Least model M_{isa}) For every well-formed *isa* hierarchy H, we call *least model of H* the set $M_{isa}(H) = M(H^{\sharp})$.

For every H, $M_{isa}(H)$ is well-defined since $M(H^{\sharp})$ does not depend on the predicate names used by the renamings used in the construction of $H^{\sharp}$. The choice of $M_{isa}(H)$ as the declarative semantics of the *isa* hierarchy H is justified by showing that the operational semantics of $H^{\sharp}$, based on SLD resolution, is equivalent to the operational semantics of H defined in terms of *isa*-proofs. The next subsection is dedicated to the proof of this result.

3.3   Equivalence between Proof Theory and Model Theory

We first show that it is not restrictive to consider *isa* hierarchies where each dynamic predicate is defined in (at most) one component. To see this we can reason as follows. Let H be the hierarchy P_n *isa* $\cdots$ *isa* P_1. For each $j \in \{1, \ldots, n\}$ build from P_j a new program $P_j^{\bullet}$ by removing from P_j, all the clauses defining dynamic predicates that are also defined in at least one of $P_n, P_{n-1}, \cdots, P_{j+1}$. Let $H^{\bullet}$ be the hierarchy obtained by replacing all the P_j's by the corresponding $P_j^{\bullet}$'s.

Lemma 3.7 For every well-formed hierarchy H, we have:

1. for every goal **G**, $H \vdash$ **G** has an *isa*-proof iff so does $H^{\bullet} \vdash$ **G**,

2. $(H^{\bullet})^{\sharp} = H^{\sharp}$ and $Pub(H^{\bullet}) = Pub((H^{\bullet})^{\sharp}) = Pub(H^{\sharp})$.

11

PROOF SKETCH. To see (1) observe (i) that each call to a dynamic predicate in H is bound to the clauses for that predicate that P_n defines or inherits from its closest ancestor in H, and (ii) that P_n and $P_n^\bullet$ contain or inherit the exact same definition for every dynamic predicate in H and $H^\bullet$.

As for (2), note that the renamings ρ_i and ϕ_i applied in the construction of $H^\sharp$ do not rename dynamic predicates. Hence, the subsetting $\bullet$ that applies to each P_i removes (at once) all and only the clauses, coming from P_i, that are removed by the iterated application of the subsetting $\star$ used in the construction of $H^\sharp$. Hence the claim. ∎

We will call DYN-disjoint every well-formed hierarchy satisfying this additional condition, ensured by the removal $\bullet$, that every dynamic predicate is defined in at most one of the components of the hierarchy. Let now $H = P_n \; isa \; \cdots \; isa \; P_1$ be a DYN-disjoint hierarchy. Then, clearly, $H^\sharp$ can be constructed simply in terms of union and renaming – hence, without subsetting – as follows:

$$(P_n \; isa \; \cdots \; isa \; P_1)^\sharp \;\; = \;\; \bigcup_{i=1}^{n} \psi_i^n(P_i).$$

Here, ψ_n^n is the identity function, and for every $i < n$, ψ_i^n is the composition of the renamings applied at the ith component of H, when constructing $H^\sharp$:

$$\psi_j^n(P_j) \;\; = \;\; (\phi_{n-1}(\rho_{n-1}(\cdots(\phi_j(\rho_j(P_j)))\cdots))).$$

Next, we show that there exists a one-to-one correspondence between the definitions for the calls to the static predicates of H and and their counterpart in $H^\sharp$.

Lemma 3.8 Let $H = P_n \; isa \; \ldots \; isa \; P_1$ be a well-formed hierarchy. For every index $j \in \{1 \ldots n\}$ and every $p \in \text{STAT}(P_j \; isa \; \ldots \; isa \; P_1)$, let $k = max\{i \le j \mid p \in Defn(P_i)\}$. Then the following properties hold true:

(a) $\psi_j^n(p) \in Defn(\psi_k^n(P_k))$,
(b) $\forall s \in \{1,\ldots,n\} \setminus \{k\}, \; \psi_j^n(p) \notin Defn(\psi_s^n(P_s))$.

PROOF. First note that the index k in the hypothesis exists for every hierarchy that satisfies the well-formedness condition. The proof is by induction on the structure of the hierarchy, i.e. the number of its components.

The base case, for hierarchies with $n = 1$ component, follows immediately by the well-formedness constraints. For the inductive case, $(n > 1)$ we proceed as follows. The inductive hypothesis on the hierarchy $P_{n-1} \; isa \; \ldots \; isa \; P_1$ guarantees that:

$$\psi_j^{(n-1)}(p) \in Defn(\psi_k^{(n-1)}(P_k)),$$
$$\forall s \in \{1,\ldots,n-1\} \setminus \{k\}, \; \psi_j^{(n-1)}(p) \notin Defn(\psi_s^{n-1}(P_s)).$$

Now consider adding P_n to this hierarchy: we have then

$$H = P_n \cup \phi_{n-1}(\rho_{n-1}(P_{n-1} \; isa \; \ldots \; isa \; P_1)^\sharp).$$

Being renamings, ϕ_{n-1} and ρ_{n-1} are injective and this, together with the inductive hypothesis tells us that for every $j < n$,

(a') $\psi_j^n(p) \in Defn(\psi_k^n(P_k))$,
(b') $\forall s \in \{1,\ldots,n-1\} \setminus \{k\}, \; \psi_j^n(p) \notin Defn(\psi_s^n(P_s))$.

12

To prove the claim for $j < n$, we need to show that (b') holds also for $s = n$, To see this, observe that $\psi_n^n(P_n) = P_n$ by definition, and hence, $\psi_n^n(p) = p$. Now, if $p \in Defn(P_n)$, then ϕ_{n-1} renames all the occurrences of $\psi_j^{n-1}(p)$ that clash with p; otherwise, if $p \notin Defn(P_n)$, $\psi_j^{n-1} = \psi_j^n$. In neither case, $\psi_j^n(p)$ belongs to $Defn(P_n)$.

Now we are left with the case $j = n$. If $p \in Defn(P_n)$, then for $j = n$ the choice of k yields $k = n$. Then (a) holds trivially whereas for (b), we need to show that:

$$\forall s \in \{1, \ldots, n-1\} p \notin Defn(\psi_s^n(P_s)).$$

But this is obvious because again, being p defined by P_n, the renaming ϕ_{n-1} renames all the occurrences of ψ_s^{n-1} that clash with p. Finally, if $p \notin Defn(P_n)$, then $\psi_n^n(p) = \psi_{n-1}^n(p)$ and we are again in the case $j < n$ that we just proved. ∎

Now we show that there exists a one-to-one correspondence between the steps of an *isa*-proof in H and the steps of a corresponding SLD derivation in $H^\sharp$. The proof uses lemma 3.8 and corresponding properties for the dynamic, extensible and internal predicates of H.

Theorem 3.9 Let $H = P_n \ isa \ldots isa \ P_1$ be a well-formed, DYN-disjoint *isa* hierarchy. Then, for every $j \in [1..n]$, every goal **G** such that $Pred(\mathbf{G}) \subseteq Pub(P_j \ isa \ \ldots \ isa \ P_1)$ and every substitution ϑ:

there exists an *isa*–proof for $j, H \vdash_\vartheta \mathbf{G}$ $\iff$ there exists a derivation $\psi_j^n(\mathbf{G}) \overset{\vartheta}{\leadsto}_{H^\sharp} \Box$.

PROOF. The proof is by induction on the height of the *isa*-proof on one side and the length of the SLD-derivation on the other side. In order to apply the induction we need a stronger hypothesis: hence we will in fact prove a stronger result and show that the thesis hold for any goal G such that $Pred(\mathbf{G}) \subseteq \text{INT}(P_j) \cup Pub(P_j \ isa \ \ldots \ isa \ P_1)$.

The base case is trivial. In the inductive step, the case when **G** is a conjunctive goal follows immediately by the inductive hypothesis, the inference rule for conjunctive goals in the *isa*-proofs, and standard properties of SLD-derivations. Let now **G** be an atom, say $p(\mathbf{t})$. We distinguish four cases, depending on the annotation for p: in all of them, it suffices to show that the first backchaining step of the *isa*-proof selects a clause in the component k if and only if the first step in the SLD-derivation selects a clause in $\psi_k^n(P_k)$.

$p \in \text{DYN}(P_j \ isa \ \ldots \ isa \ P_1)$. We already noted that the dynamic predicates are not renamed in the construction of $H^\sharp$: consequently, $\psi_j^n(p(\mathbf{t})) = p(\mathbf{t})$. Furthermore, since H is DYN-disjoint, there exists at most one $k, 1 \le k \le n$ such that $p \in Defn(P_k)$. If such a k exists, then the first step of the *isa*-proof selects a clause in P_k, say $p(\mathbf{s}) \leftarrow \mathbf{G}$, such that $\gamma = mgu(p(\mathbf{s}), p(\mathbf{t}))$, and solve $\mathbf{B}\gamma$ from the k-th component of H. Correspondingly, the first step of the SLD-derivation in $H^\sharp$ will produce the resolvent $\psi_k^n(\mathbf{B}\theta)$ where $Pred(\mathbf{B}\gamma) \subseteq Preds(P_k)$. Now, the claim follows by the inductive hypothesis.

$p \in \text{EXT}(P_j \ isa \ \ldots \ isa \ P_1)$. This case differs from the previous one only because it may exists more than one component P_k containing a definition for $\psi_j^n(p(\mathbf{t})) = p(\mathbf{t})$. Since any clause in these components can be non-deterministically selected both by the *isa*-proof and by the SLD-derivation, the result follows as in the previous case.

$p \in \text{INT}(P_j)$. Since H is well-formed, the renamings ρ never rename the internal predicates of the components, P_is of H: consequently, $\psi_j^n(p(\mathbf{t})) = p(\mathbf{t})$. Note also that none of the *new* internal predicates of H (i.e. those deriving from the renaming of the static predicates) may

clash with p. Therefore the first step of the SLD-derivation in $H^\sharp$ will select a clause in $\psi_j^n(P_j)$ if and only if the same is true of the first step in the *isa*-proof. Again, the result follows immediately from the inductive hypothesis.

$p \in \text{STAT}(P_j \ isa \ \ldots \ isa \ P_1)$. This is the only non-straightforward case, but the proof goes through as in the previous cases using the result of lemma 3.8. ∎

The connection between a proof in an *isa* hierarchy H and its corresponding SLD refutation in $H^\sharp$ can be finally established as follows.

Theorem 3.10 (Equivalence) Let H be a well-formed *isa* hierarchy. Then, for every $\mathbf{G}$ such that $Pred(\mathbf{G}) \subseteq Pub(H)$,

$$\text{there exists an } isa\text{-proof for } H \vdash \mathbf{G} \quad \Longleftrightarrow \quad \mathbf{G} \text{ is refutable in } H^\sharp.$$

PROOF. It is not restrictive to assume that H is DYN-disjoint and that $Pub(H) = Pub(H^\sharp)$. In fact, by Lemma 3.7, for every hierarchy H there always exists a DYN-disjoint hierarchy which behaves exactly as H on every goal and whose public predicates coincide with those of $H^\sharp$. Now, when $Pred(\mathbf{G}) \subseteq Pub(H)$, there exists an *isa*-proof for $H \vdash \mathbf{G}$ if and only if there exists a substitution ϑ such that $n, H \vdash_\vartheta \mathbf{G}$ has an *isa*-proof. By Theorem 3.9, this is true if and only if $\psi_n^n(\mathbf{G}) \overset{\vartheta}{\leadsto}_{H^\sharp} \square$ and the claim follows because $\psi_n^n(\mathbf{G}) = \mathbf{G}$. ∎

Owing to this equivalence, we can take $M_{isa}(H)$ as the declarative semantics of every hierarchy H as we show in the next theorem.

Theorem 3.11 (Soundness and completeness) For every well-formed *isa* hierarchy H, and every goal $\mathbf{G}$:

$$\text{there exists an } isa\text{-proof for } H \vdash \mathbf{G} \quad \Longleftrightarrow \quad \text{there exists } \vartheta \text{ such that } M_{isa}(H) \models \mathbf{G}\vartheta.$$

PROOF. Again we can assume that H is DYN-disjoint and that $Pub(H) = Pub(H^\sharp)$. Observe that the existence of a proof for $H \vdash \mathbf{G}$ and the satisfaction of $\mathbf{G}\vartheta$ in $M(H^\sharp)$ both imply that $Pred(\mathbf{G}) \subseteq Pub(H^\sharp)$. Hence $M_{H^\sharp} \models \mathbf{G}\vartheta$ iff $M(H^\sharp) \models \mathbf{G}\vartheta$ because the projection involved in $M(H^\sharp)$ filters out only non-public predicates. Now, since by definition $M(H^\sharp) \models \mathbf{G}\vartheta$ iff $M_{isa}(H) \models \mathbf{G}\vartheta$, the proof of both the claims is immediate from Theorem 3.10 using the classical soundness and completeness result for SLD derivations [18]. ∎

3.4 Model Theory and Computational Behavior

The equivalence between the proof theory and the model theory stated in Theorem 3.11 is important in that it shows that computing with an *isa* hierarchy of modules is equivalent to applying logical inferences, with SLD resolution, in a corresponding logic program. In this respect, Theorem 3.11 gives logical foundations to the composition operator we have introduced. However, it does not give any insight as to whether the model theory provides indeed an adequate characterization of the computational properties of this operator.

Clearly we can use the model theoretic semantics to reason about hierarchies of programs. For instance, we can tell whether a goal G is provable in a hierarchy H by testing that G is satisfiable in the least model $M_{isa}(H)$. Similarly, as we show next, we can tell whether two hierarchies are equivalent by looking at their least models.

Definition 3.12 Two hierarchies H_1 and H_2 are equivalent (written $H_1 \equiv_{isa} H_2$) if and only if they prove exactly the same goals, i.e. if and only if for every goal **G**,

$$H_1 \vdash \mathbf{G} \text{ has an } isa \text{ proof} \iff H_2 \vdash \mathbf{G} \text{ has an } isa \text{ proof.}$$

Proposition 3.13 Two well-formed hierarchies H_1 and H_2 are equivalent if and only if their least models coincide, i.e.:

$$H_1 \equiv_{isa} H_2 \iff M_{isa}(H_1) = M_{isa}(H_2).$$

PROOF. From Theorem 3.11 it follows that for every ground goal **G**,

$$H \vdash \mathbf{G} \text{ has an } isa\text{-proof} \iff M_{isa}(H) \models \mathbf{G}. \tag{1}$$

From (1) the implication ($\Rightarrow$) follows from Definition 3.12 because for every ground atom A, $A \in M_{isa}(H)$ iff $M_{isa}(H) \models A$.

The proof of the converse implication is by contradiction. Assume that there exists a goal **G** that is provable in H_1 and not in H_2. Then, since $H_1 \vdash \mathbf{G}$ has an isa-proof, there exists a substitution β, grounding for **G**, such that $H \vdash \mathbf{G}\beta$ has also an isa proof. On the other hand, since $H_2 \vdash \mathbf{G}$ has no isa-proof, it follows that $H_2 \vdash \mathbf{G}\beta$ has no isa-proof either. This together with (1) contradicts the hypothesis $M_{isa}(H_1) = M_{isa}(H_2)$. ∎

Reasoning on the equivalence between hierarchies is important in several situations. Consider for instance removing a module from a hierarchy or replacing that module with a different one. Clearly, in situations like these, we would like to be able to tell whether the computational behavior of the original hierarchy has been preserved through the transformation. However, the typical operations that characterize a modular language are operations on the modules of a program rather than on the program itself. Separate compilation or analysis are examples of these operations. For instance we might want to apply certain optimizing transformations on a module that is shared by several hierarchies, or to reason on the data flow, or control flow within a module, independently of the hierarchies that module belongs to.

Clearly, to give a formal account of these operations, we need to be able to reason on the relation of equivalence over modules, and in general on the computational behavior of a module independently of the hierarchies where a module occurs. Semantically speaking, this means that we need a notion of computational equivalence over the components of our hierarchies as well as an abstract characterization of the semantics of these components. In fact we already have the former, because we can adopt the following equivalence relation induced by the proof theory. Let $H[P]$ denote an isa hierarchy that has P as one of its component programs, and let $H[Q]$ be obtained from $H[P]$ by replacing P with Q.

Definition 3.14 ($\cong_{isa}$) We say that two differential programs P and Q are isa-equivalent, and we write $P \cong_{isa} Q$, if and only if for every hierarchy H such that $H[P]$ and $H[Q]$ are well-formed

$$H[P] \equiv_{isa} H[Q].$$

Strictly speaking, $\cong_{isa}$ is a congruence: we are saying that two modules are equivalent, or congruent, under $\cong_{isa}$ if and only if they behave indistinguishably in every hierarchy they may belong to. Clearly, as stated, the definition has little practical significance, because to prove two programs $\cong_{isa}$-congruent it requires an equivalence test (under $\equiv_{isa}$) for all the possible hierarchies that may be formed out of the two programs. Yet, if we can define a semantics for

the components of a hierarchy, and this semantics is *adequate*, in the sense we explain next, then we can reason on this computational equivalence in terms of semantic equality.

The notion of adequacy we employ is standard: we say that a semantics is *correct* if and only if semantic equality implies computational equivalence: we say that the semantics is *fully abstract* if semantic equality and computational equivalence coincide.

When instantiated to the case of differential logic programs, these definitions are as follows. Denote with $[\![\cdot]\!]$ a semantics over differential programs – a function from the class of differential programs to some (yet to be specified) semantic domain. Then, we say that $[\![\cdot]\!]$ is $\cong_{isa}$-correct if and only if for any two differential programs P and Q,

$$[\![P]\!] = [\![Q]\!] \ \Rightarrow \ P \cong_{isa} Q.$$

We say that $[\![\cdot]\!]$ is *fully abstract* wrt $\cong_{isa}$ if and only if

$$[\![P]\!] = [\![Q]\!] \ \Leftrightarrow \ P \cong_{isa} Q.$$

Correctness is clearly a must for an adequate semantics. Full abstraction, in its turn, is just as important, for it guarantees that any distinction made at the semantic level has also a computational counterpart. Note that, as shown by Proposition 3.13, the least model provides a fully abstract semantics wrt $\equiv_{isa}$. The study of a semantics fully abstract wrt $\cong_{isa}$ for our programs is carried out in detail in the next section. We first define a compositional semantics for the operator $\lhd$ on differential programs and then we prove this semantics $\cong_{isa}$-correct. We conclude proving that it is fully abstract.

4   A Compositional and Fully Abstract Semantics

In order to characterize the behavior of a program within a hierarchy, we clearly need a functional semantics because we need a semantic account of the potential interactions of that program with the other components of the hierarchy: given an interpretation of the hierarchy where the program occurs, the semantics of the program is a function of that interpretation. Functional semantics for logic program have been already investigated in the literature: the T_P operator of [19] and the closure operator $(T_P + id)^\omega$ [17] are examples of such semantics.

The approach we follow here is inspired by the work on of *open* logic programs developed in [7] and [8]. For the ground case, the semantics for *open* logic programs is in several respects similar to the semantics based on $(T_P + id)^\omega$ of [17], but it refines it in that it provides a syntactic representation of the closure operator, a kind of normal form representation for $(T_P + id)^\omega$.

Like differential programs, *open* logic programs can be composed into collections that are obtained taking the union of the components' clauses. In this respect, an *open* logic program can be seen as a differential program which uses only extensible (and internal, for that matter) predicate symbols. When viewed as part of a composition, the behavior of an open program changes depending on the definitions of the extensible predicates that come from the other components. Semantically, this dependency is modeled by defining the meaning of the program program as (an abstract representation of) the set of partial SLD derivations that end up in resolvent that are constituted solely of extensible predicates.

Denoting with Ω the set of extensible (i.e. open) predicates of an open program, in [7] the ground semantics of the program is defined as follows. Given a set of clauses I, denote with

$Ground(I)$ the set of all the ground instances (on the given alphabet) of the clauses of I.

$$O_\Omega(P) \quad = \quad Ground(\{A \leftarrow \mathbf{B} \mid \quad A \text{ is an atom, there exists a derivation } A \stackrel{\vartheta}{\rightsquigarrow}_{P,R} \mathbf{B}$$
$$\text{and } Pred(\mathbf{B}) \subseteq \Omega\}).$$

Given the notion of observables we have adopted, in [7], an abstraction of this semantics is proved correct and fully abstract with respect to the relation of computational equivalence induced by the union of open program. Our semantics of differential programs is defined along similar guidelines, but with important differences that we next illustrate and motivate.

Let's assume, for the moment, that O_Ω "does the right thing" also in our case. The assumption seems justified because, as for an open logic program, the behavior of a differential program P depends on the definitions of the public predicates of P that come from the components of a hierarchy. More precisely, the behavior of P depends on the predicates in the set $Open(P) = Pub(P) \setminus (\text{STAT}(P) \cap Defn(P))$, because the remaining (static and internal) predicates of P are evaluated locally to the program, disregarding the context where P occurs.

Consider then the two differential programs $P = \{\texttt{q(a).}, \ \texttt{p(X)} \leftarrow \texttt{q(X).}\}$ and $Q = \{\texttt{q(b).}\}$, where p and q are both *dynamic* predicates. Being p and q dynamic, we can take $\Omega = Open(P) = \{p, q\}$ and apply the definition of O_Ω to obtain:

$$O_\Omega(P) \quad = \quad \{q(a), \ p(a), \ p(a) \leftarrow q(a), \ p(b) \leftarrow q(b)\},$$
$$O_\Omega(Q) \quad = \quad \{q(b)\}.$$

Now consider the hierarchy $Q \ isa \ P$ and compute the corresponding program $(Q \ isa \ P)^\sharp$. Applying the definition we have:

$$(Q \ isa \ P)^\sharp = Q \lhd P = \{\texttt{q(b).}, \ \texttt{p(X)} \leftarrow \texttt{q(X).}\}$$

because the unit clause $\texttt{q(a)}$ of P gets overridden by the corresponding clause $\texttt{q(b)}$ of Q. Now, it is clear that we cannot obtain the meaning of $Q \ isa \ P$ directly from $O_\Omega(P)$ and $O_\Omega(Q)$. Note, in this regard, that neither $p(a)$ nor $q(a)$ have a proof in $Q \ isa \ P$ according to our proof theory. Hence, we would expect something like the following:

$$O_\Omega(Q \ isa \ P) = \{p(b), \ q(b), \ p(a) \leftarrow q(a), \ p(b) \leftarrow q(b)\}$$

as the meaning of $Q \ isa \ P$. Now, in order to get $O_\Omega(Q \ isa \ P)$ from $O_\Omega(P)$ and $O_\Omega(Q)$ we should delete not only $q(a)$, as we expect as a consequence of the overriding semantics of *isa* but also $p(a)$ which is derived from $q(a)$. But to do so, when we define the semantics of P, we need a mechanism for recording that $p(a)$ has been obtained using the definition of the dynamic predicate q, local to P, which will be overridden in the hierarchy $Q \ isa \ P$.

The following notion of *context sensitive clause* helps formalize this idea.

Definition 4.1 A *context sensitive clause (cs-clause)* is an object of the form

$$A \leftarrow \{q_1, \ldots, q_n\} \ \square \ B_1, \ldots, B_k$$

where $q_1, \ldots, q_n$ are predicate symbols and $A, B_1, \ldots, B_k$ are atoms. We say that $\{q_1, \ldots, q_n\}$ and $B_1, \ldots, B_k$ are respectively the *context* and the (proper) *body* of the cs-clause.

17

The intuitive interpretation of the cs-clause $A \leftarrow s \;\square\; \mathbf{B}$ is that the logical implication $A \leftarrow \mathbf{B}$ is true in every context (or hierarchy) which does not override the definition of the predicates in s. Accordingly, every clause can be seen as a cs-clause with an empty context: simply, it states the truth of an implication independently of any context. We will henceforth adopt this interpretation and, to ease the notation, we will use the same notation for cs-clauses with empty context and clauses.

The semantic domain for interpreting our programs is defined in terms of a corresponding notion of *context sensitive interpretation*. Here we restrict the semantic domain to ground objects: besides being consistent with the notion of observables we have chosen in section 3.1, this restriction allows us to view the proper body of a cs-clause as a set of atoms and to rely on the same the technical simplifications of the ground semantics in [7].

Definition 4.2 (cs-interpretation) Let C denote the set of all the ground cs-clauses whose proper bodies are considered as sets. A *context sensitive* interpretation (cs-interpretation) is any $I \subseteq C$.

The semantics of a differential program is defined by a fixed point construction based on the immediate-consequence operator D_P over *cs*-interpretations that we first introduce and then explain. Given any set of predicate symbols Ω, we denote by Id_Ω the set of tautological cs-clauses:

$$\mathrm{Id}_\Omega \;\; = \;\; \{ p(\mathbf{t}) \leftarrow p(\mathbf{t}) \mid p \in \Omega \text{ and } \mathbf{t} \text{ is a tuple of ground terms} \}.$$

Definition 4.3 (D_P) For every differential program P, the immediate consequence operator $D_P : \wp(C) \mapsto \wp(C)$ is the function defined as follows. Let I be an arbitrary cs-interpretation and let $Open(P)$ denote the set $Pub(P) \setminus (\mathrm{STAT}(P) \cap Defn(P))$ of the *open* predicates of P.

$$
\begin{aligned}
D_P(I) \;\; = \;\; & \{ A \leftarrow s_0 \cup s_1 \cdots \cup s_k \;\square\; \mathbf{L}_1, \ldots, \mathbf{L}_k \in \mathcal{C} \mid \\
& \quad \text{there exists } A \leftarrow B_1, \ldots, B_k \in Ground(P), \\
& \quad \text{for every } i \in [1..k] \text{ there exists } B_i \leftarrow s_i \;\square\; \mathbf{L}_i \in I \cup \mathrm{Id}_{Open(P)}, \\
& \quad s_0 = \{ Pred(B_i) \mid Pred(B_i) \subseteq \mathrm{DYN}(P) \text{ and } B_i \leftarrow s_i \;\square\; \mathbf{L}_i \notin \mathrm{Id}_{Open(P)} \} \}.
\end{aligned}
$$

Applying D_P to an interpretation I corresponds to perform one step of unfolding on all the body atoms of the clauses of the program, using the cs-clause contained in $I \cup \mathrm{Id}_{Open(P)}$. As in [7], the intention is to model with $D_P \uparrow \omega$ – the interpretation obtained by the iterated application of D_P starting from the empty interpretation – the set of partial derivations of P ending up in open predicates. Unfolding an atom with a tautological clause in the set $\mathrm{Id}_{Open(P)}$ corresponds to a null operation that mimics the effect of delaying the selection of that atom in a corresponding SLD derivation. On the other hand, unfolding an atom with a cs-clause from I corresponds to selecting that atom in the derivation. For static, extensible and internal predicates unfolding does nothing special: it simply replaces the atom with the body (including the context) of the cs-clause used to unfold the atom. Dynamic predicates have a special status because unfolding one such atom adds the predicate symbol of atom to the context of the resulting cs-clause. This way, we model the dependency of the corresponding derivation on the definition of the dynamic predicate, which might get overridden when the program is composed into a hierarchy.

Proposition 4.4 (Continuity) D_P is continuous on the complete lattice $(\wp(\mathcal{C}), \subseteq)$.
PROOF. Standard. ∎

Since D_P is continuous, $D_P \uparrow \omega$ is the least fixed point of D_P. Every cs-clause $A \leftarrow s \;\square\; \mathbf{B}$ in the fixed point represents (a ground instance of) a partial derivation in P, from the head A

of the cs-clause to the resolvent $\mathbf{B}$. Furthermore, being $D_P \uparrow \omega$ closed under unfolding, all the predicate symbols in $Pred(\mathbf{B})$ are *open* because the set $\mathtt{Id}_{Open(P)}$ contains only the tautological clauses that are relative to the open predicates of P. The context s, finally, holds the symbols of the dynamic predicates used in all the unfolding steps of the partial derivation.

The semantics of a program P is defined as an abstraction of $D_P \uparrow \omega$ that we motivate first, and then define formally. There are three levels at which $D_P \uparrow \omega$ can be abstracted upon, reasoning on the structure of the cs-clauses contained in $D_P \uparrow \omega$. Let $\mathtt{C}: \; A \leftarrow s \, \square \, \mathbf{B}$ be one such clause.

Hiding of internal predicates

Consider first the case when $Pred(A) \subseteq \mathrm{INT}(P)$. Since the definitions of the internal predicates of P are private to the program, and since $D_P \uparrow \omega$ is closed under unfolding (being $D_P \uparrow \omega$ a fixed point) the cs-clauses defining these predicates in $D_P \uparrow \omega$ can be dispensed with and removed altogether. This removal mimics the fact that the internal predicates are not exported by the program as it happens in the proof theory.

Abstraction based on dynamic predicates

Suppose first that $Pred(A) = \{p\} \subseteq s$. That p is a dynamic predicate follows directly from the definition of D_P because the context s in $\mathtt{C}$ holds the symbols of all the dynamic predicate used in the derivation from A to $\mathbf{B}$. But clearly, since the derivation associated to $\mathtt{C}$ starts off with the predicate in A, the dependency of $\mathtt{C}$ on $Pred(A)$ is encoded by the head of $\mathtt{C}$. Hence, $Pred(A)$ needs not be included in s and we can replace $\mathtt{C}$ in $D_P \uparrow \omega$ with the new cs-clause $A \leftarrow s \setminus \{Pred(A)\} \, \square \, \mathbf{B}$ without loss of generality.

Now consider the case when the body $\mathbf{B}$ contains an atom B whose predicate, say p, is dynamic. Then, if $Pred(A) = Pred(B)$ or $Pred(B) \subseteq s$, we can safely remove $\mathtt{C}$ from $D_P \uparrow \omega$. To see that, we can reason as follows. First note that in both cases p is defined locally to P: this is obvious when $Pred(A) = Pred(B)$, whereas in the other case, it follows by observing that $Pred(B) \subseteq s$ implies that $\mathtt{C}$ is the result of at least one step of unfolding that uses a definition for p. Let then $H[P]$ be an arbitrary *isa* hierarchy containing P. Since p is defined locally to P, the calls to p in P do not depend on the definitions coming from the ancestors of P in $H[P]$. On the other hand, if we look at the heirs of P, we can distinguish two situations. If none of the heirs defines p, then a call to p in P will always be unfolded using the clauses of the definition of p local to P. On the contrary, if one of the heirs defines p, then none of these clauses will ever be selected. Now, since $D_P \uparrow \omega$ is closed under unfolding, all of the unfoldings on $Pred(B)$ that use clauses local to P are encoded in the cs-clauses of $D_P \uparrow \omega$. Hence, we can remove from $D_P \uparrow \omega$ all the cs-clauses that encode partial derivations that have selected p at least once and that would need to select p again to terminate. These are precisely the cs-clauses where $Pred(A) = Pred(B)$, because these clauses encode derivations that start off with p, and the cs-clauses where $p \in s$.

Abstraction based on Subsumption equivalence

To introduce the third, and last, abstraction, we first need the following definition.

Definition 4.5 We say that the cs-clause $A \leftarrow s \, \square \, B_1, \ldots, B_n$ is a *tautology* if and only if there exists $i \in [1..n]$ such that $B_i = A$. Given two ground cs-clauses $\mathtt{C}_1: \; A \leftarrow s_1 \, \square \, B_1, \ldots, B_n$ and $\mathtt{C}_2: \; A \leftarrow s_2 \, \square \, D_1, \ldots, D_m$, we say that $\mathtt{C}_1$ *subsumes* $\mathtt{C}_2$ if and only if $\{B_1, \ldots, B_n\} \subseteq$

$\{D_1, \ldots, D_m\}$ and $s_1 \subseteq s_2$. Finally, we say that $\texttt{C}_1$ subsumes $\texttt{C}_2$ *strictly* if and only if $\texttt{C}_1$ subsumes $\texttt{C}_2$ and $\texttt{C}_2$ does not subsume $\texttt{C}_1$.

With the third abstraction we remove from $D_P \!\uparrow\! \omega$ all the tautologies as well as all the cs-clauses that are strictly subsumed by other cs-clauses in $D_P \!\uparrow\! \omega$. Following the standard terminology, we say that set resulting from this abstraction is the *weak canonical form* of the quotient of $D_P \!\uparrow\! \omega$ under subsumption equivalence. The correctness of this abstraction is motivated as follows.

Consider first the case of a tautology. If the predicate of the head of the tautology is a dynamic, we can reason as in the previous case. Otherwise, the tautology encodes an infinite derivation whose contribution to the semantics of the program is null.

Now assume that $D_P \!\uparrow\! \omega$ contains two cs-clauses $\texttt{C}_1$ and $\texttt{C}_2$ such that $\texttt{C}_1$ strictly subsumes $\texttt{C}_2$. Consider first the case when $\texttt{C}_1$ and $\texttt{C}_2$ differ only because the context s_1 of $\texttt{C}_1$ is a subset of the context s_2 of $\texttt{C}_2$. Let, for instance, s_1 be the empty set and $s_2 = \{q\}$. Now $\texttt{C}_1$ and $\texttt{C}_2$ bear the exact same meaning as derivations: the only difference is that $\texttt{C}_2$ encodes a derivation that is meaningless in every context that redefines q, whereas this is not true of the derivation encoded by $\texttt{C}_1$. Hence, when computing the semantics of P we can safely drop $\texttt{C}_2$ as long as we retain $\texttt{C}_1$. Consider then the case when the body of $\texttt{C}_1$ is contained in the body of $\texttt{C}_2$ and let A be the head of the two cs-clauses. Being the body of $\texttt{C}_1$ contained in the body of $\texttt{C}_2$, clearly the derivation encoded by $\texttt{C}_1$ leads to a proof of the head A whenever so does the derivation encoded by $\texttt{C}_2$. Hence, we can safely remove $\texttt{C}_2$ as long as we have $\texttt{C}_1$.

The three abstraction we have discussed are implemented by the three functions over cs-interpretations introduced in the following definition.

Definition 4.6 (α) Let P be a differential program and I be a cs-interpretation. The abstraction $\alpha(I)$ is defined as follows:
$$\alpha(I) = wcf(\delta(\iota(I)))$$

where
$$\iota(I) = \{A \leftarrow s \,\square\, \mathbf{B} \in I \mid Pred(A) \not\subseteq \text{INT}\},$$

$$\delta(I) = \{A \leftarrow s \,\square\, \mathbf{B} \mid \quad \text{there exists } A \leftarrow s' \,\square\, \mathbf{B} \in I \text{ such that}$$
$$s = s' \setminus Pred(A), \ s \cap Pred(\mathbf{B}) = \emptyset \text{ and}$$
$$Pred(A) \not\subseteq (Pred(\mathbf{B}) \cap \text{DYN})\},$$

$$wcf(I) = \{\texttt{C} \in I \mid \quad \texttt{C} \text{ is not a tautology and there exists}$$
$$\text{no } \texttt{C}' \in I \text{ such that } \texttt{C'} \text{ strictly subsumes } \texttt{C}\}.$$

The semantics of a differential program P is defined in terms of the α-abstraction of the fixed point of D_P. In order to account for overriding semantically, we need also to encode information about those predicates of the program whose definitions are subject to overriding: clearly, these are the static and dynamic predicates that are defined by the program. Let then ∇_P denote the set $\nabla_P = (\text{STAT} \cup \text{DYN}) \cap Defn(P)$.

Definition 4.7 ($[\![\cdot]\!]$) Given the differential program P, let $F(P)$ denote the α-abstraction of $D_P \!\uparrow\! \omega$, i.e. $F(P) = \alpha_P(D_P \!\uparrow\! \omega)$. The semantics $[\![P]\!]$ of P is defined as the pair:

$$[\![P]\!] \ = \ (F(P), \nabla_P).$$

The remaining of this section is dedicated to the proof that $[\![\cdot]\!]$ is a correct and fully abstract semantics with respect to the computational equivalence $\cong_{isa}$ introduced in Section 3.2. We first show that $[\![\cdot]\!]$ is $\lhd$-compositional. Based on this result we show that $[\![\cdot]\!]$ is $\cong_{isa}$-correct and then we conclude proving that $[\![\cdot]\!]$ is fully abstract.

4.1 Correctness and Full Abstraction

We start introducing a composition operator for the semantics of two programs that gives a semantic account of the syntactic $\lhd$-composition of the programs.

Definition 4.8 Let P and Q be differential programs such that P *isa* Q is a well-formed hierarchy. Denote with $F_r(Q)$ and $F_l(P)$ the subsets of, respectively, $F(Q)$ and $F(P)$ defined as follows:

$$
\begin{aligned}
F_r(Q) &= \{A \leftarrow s \,\square\, \mathbf{B} \in F(Q) \mid s \cap \nabla_P = \emptyset \text{ and } Pred(A) \not\subseteq \nabla_P\}, \\
F_l(P) &= \{A \leftarrow s \,\square\, \mathbf{B} \in F(P) \mid Pred(\mathbf{B}) \cap \text{STAT} \subseteq Defn(F(Q))\}.
\end{aligned}
$$

The semantic composition $\ll$ of $[\![P]\!]$ and $[\![Q]\!]$ is defined as follows:

$$[\![P]\!] \ll [\![Q]\!] = (F(P) \prec F(Q), \nabla_P \cup \nabla_Q)$$

where $F(P) \prec F(Q)$ is the cs-interpretation obtained from $F(P)$ and $F(Q)$ as defined below:

$$F(P) \prec F(Q) = F(F_l(P) \cup F_r(Q)).$$

Remark. As stated, the notation $F(P) \prec F(Q)$ is loosely defined because D_P, and consequently $F(P)$, are defined on differential programs whereas $F_l(P) \cup F_r(Q)$ is a cs-interpretation. However, it is immediate to see how D_P (and $F(P)$) can be extended to the case when P is a set of cs-clauses rather than simply a set of clauses as it is done in Definition 4.3.

The intuitive reading of Definition 4.8 is as follows. Consider composing the two programs P and Q in the hierarchy P *isa* Q or, equivalently, in the program $P \lhd Q$.

The subsetting on $F(Q)$ models the overriding that affects the definitions of Q in the composition. Consider a predicate symbol $p \in \nabla_Q$ and assume that p is static. Being closed under unfolding, $F(Q)$ encodes all the local selections of the static predicates that are defined in Q. Thus, by removing the cs-clauses that still define p in the fixed point, the subsetting $F_r(Q)$ realizes the combination of static binding and overriding for these predicates. On the other hand, if p is dynamic, the combination of overriding and dynamic binding is realized by the removal of all the cs-clauses in $F(Q)$ that encode derivations that use the local definitions of p, i.e. of all the derivations that start off with p (hence the condition $Pred(A) \not\subseteq \nabla_P$), and the derivations that use a local definition of that predicate (whence $s \cap \nabla_P = \emptyset$).

To motivate the subsetting $F_l(P)$, first observe that if the body of a cs-clause in $F(P)$ contains an atom whose predicate is static, then, according to the definition of D_P, that predicate is not defined by P. Now, being P *isa* Q well-formed by hypothesis, all of the static predicates of P that are not defined locally to P must be defined in Q. But then, in the differential program $P \lhd Q$, these predicates are not "open", and if there is no cs-clause defining one such predicate in $F(Q)$, then all of the derivations that start off in that predicate fail (locally) in Q and hence in $P \lhd Q$. Hence, we can remove all of the cs-clauses that encode partial derivations that, to be completed, would need further selections of any such predicate.

The proof that $[\![\cdot]\!]$ is $\lhd$-compositional uses the following two lemmas whose proofs are given in the Appendix.

Lemma 4.9 Let P and Q be differential programs such that $P\,isa\,Q$ is a well-formed hierarchy. Then

$$F(\phi(Q^\star)) = F_r(Q)$$

where $\phi(Q^\star)$ and $F_r(Q)$ are defined wrt to P, according to Definitions 3.3 and 4.8, respectively.

Lemma 4.10 Let P and Q be differential programs such that $P\,isa\,Q$ is a well-formed hierarchy and $\nabla_P \cap Defn(Q) = \emptyset$. Then:

$$F(P \cup Q) \quad = \quad F(F_l(P) \cup F(Q)).$$

Theorem 4.11 ($\lhd$-compositionality) For every two differential programs P and Q such that $P\,isa\,Q$ is a well-formed hierarchy

$$[\![P \lhd Q]\!] \quad = \quad [\![P]\!] \ll [\![Q]\!].$$

PROOF. Let $P \lhd Q$ be the differential programs associated to $P\,isa\,Q$. By definition, $P \lhd Q = P \cup \phi(Q^\star)$ and the renaming ρ and the subsetting $\star$ guarantee that $\nabla_P \cap Defn(\phi(Q^\star)) = \emptyset$. Now we can show that $F(P \lhd Q) = F(P) \prec F(Q)$:

$$
\begin{aligned}
\mathcal{F}(P \lhd Q) &= && \text{by definition of } \lhd \\
\mathcal{F}(P \cup \phi(Q^\star)) &= && \text{by Lemma 4.10 being } \nabla_P \cap Defn(\phi(Q^\star)) = \emptyset \\
\mathcal{F}(\mathcal{F}_l(P) \cup \mathcal{F}(\phi(Q^\star))) &= && \text{by Lemma 4.9} \\
\mathcal{F}(\mathcal{F}_l(P) \cup \mathcal{F}_r(Q)) &= && \text{by Definition 4.8} \\
\mathcal{F}(P) \prec \mathcal{F}(Q).
\end{aligned}
$$

To conclude the proof, observe that for every P and Q such that $P\,isa\,Q$ is well-formed, the following is an indentity: $\nabla_{P \lhd Q} = \nabla_P \cup \nabla_Q$. ∎

Theorem 4.12 (Correctness) Let P and Q be differential programs. Then

$$[\![P]\!] = [\![Q]\!] \quad \Rightarrow \quad P \cong_{isa} Q.$$

PROOF. Given a cs-interpretation I, denote by $hd(I)$ the following set of atoms:

$$hd(I) = \{A \mid A \leftarrow s \,\square \in I\}.$$

We first need the following observation.

Claim 4.13 For every differential program P, $hd(\mathcal{F}(P)) = M(P)$.
PROOF. Let T_P denote the (classical) immediate consequences operator of P. Given a cs-interpretation I, Definition 4.3 implies that $hd(D_P(I)) = T_P(hd(I))$. From this identity, it is easy to see, by induction, that for every n, $hd(D_P \uparrow n) = T_P \uparrow n$, and consequently, that:

$$hd(D_P \uparrow \omega) \quad = \quad T_P \uparrow \omega. \tag{1}$$

Now, since the abstraction ι, in $\alpha(I)$, removes from I all the unit cs-clauses whose head's predicate symbol is internal, from Definition 4.7 we have:

$$A \in hd(\mathcal{F}(P)) \quad \Longleftrightarrow \quad A \in hd(D_P \uparrow \omega) \text{ and } Pred(A) \not\subseteq \mathrm{INT}(P). \tag{2}$$

On the other hand, by definition of the least model M, we have:

$$A \in M(P) \quad \text{iff} \quad A \in T_P \uparrow \omega \text{ and } Pred(A) \not\subseteq \mathrm{INT}(P)$$

and this, together with (1) and (2) proves the claim. ∎

Now consider an arbitrary well-formed hierarchy $H[P] = R_n \; isa \; \cdots \; P \; \cdots \; isa \; R_1$ and let j be the index of P in $H[P]$. Denote by $H[Q]$ the hierarchy obtained from $H[P]$ by replacing P with Q and assume that both $H[P]$ and $H[Q]$ are well-formed. Let finally ψ_i^n be the composition of the renamings applied to the ith component of $H[P]$ (and $H[Q]$) in the construction of $H[P]^\sharp$ ($H[Q]^\sharp$) as defined in section 3.2.

Since the reamings ρ used in Definition 3.4 rename only internal predicates and the clauses defining internal predicates are deleted by the abstraction α, clearly for any R

$$F(\rho_j(R)) = F(R) \tag{3}$$

holds. We can now conclude the proof by showing that $M_{isa}(H[P]) = M_{isa}(H[Q])$.

$$
\begin{aligned}
M_{isa}(H[P]) &= \text{by Definition 3.6} \\
M((R_n \; isa \ldots P \ldots isa \; R_1)^\sharp) &= \text{by Definition 3.4} \\
M(R_n \lhd \rho_n((R_{n-1} \; isa \ldots P \ldots isa \; R_1)^\sharp)) &= \text{by Claim 4.13} \\
hd(F(R_n \lhd \rho_n((R_{n-1} \; isa \ldots P \ldots isa \; R_1)^\sharp))) &= \text{by Theorem 4.11} \\
hd(F(R_n) \prec F(\rho_n((R_{n-1} \; isa \ldots P \ldots isa \; R_1)^\sharp))) &= \text{by (3)} \\
hd(F(R_n) \prec F((R_{n-1} \; isa \ldots P \ldots isa \; R_1)^\sharp)) &= \text{repeating the previous steps} \\
&\vdots \\
hd(F(R_n) \prec F(R_{n-1}) \ldots \prec F(P) \prec F((R_{i-1} \; isa \ldots \; R_1)^\sharp)) &= \text{being } \llbracket P \rrbracket = \llbracket Q \rrbracket \\
hd(F(R_n) \prec F(R_{n-1}) \ldots \prec F(Q) \prec F((R_{i-1} \; isa \ldots \; R_1)^\sharp)) &= \text{reversing the previous steps} \\
&\vdots \\
M_{isa}(H[Q]). &
\end{aligned}
$$

$\blacksquare$

Theorem 4.14 (Full abstraction) Let P and Q be differential programs. Then

$$P \cong_{isa} Q \quad \Leftrightarrow \quad \llbracket P \rrbracket = \llbracket Q \rrbracket.$$

PROOF. We already proved the ($\Leftarrow$) direction in Theorem 4.12. For the ($\Rightarrow$) direction, we prove the contrapositive, showing that whenever $\llbracket P \rrbracket \neq \llbracket Q \rrbracket$ we can find a well-formed hierarchy H such that $H[P] \not\cong_{isa} H[Q]$. There are two possible reasons why $\llbracket P \rrbracket \neq \llbracket Q \rrbracket$. Consider first the case when $\nabla_P \neq \nabla_Q$ and assume, without loss of generality, that there exists $p \in \nabla_P \setminus \nabla_Q$. Let A be an atom such that $Pred(A) = \{p\}$. We distinguish two subcases depending on whether A has a refutation in P or not.

    If A has a refutation in P, then take the two hierarchies $H[P] = P \; isa \; \mathtt{T}$ and $H[Q] = Q \; isa \; \mathtt{T}$ where $\mathtt{T}$ is a set of tautological clauses for the static predicates of P and Q that make $H[P]$ and $H[Q]$ both well-formed. Then clearly, $H[P] \vdash A$ has an isa-proof whereas $H[Q] \vdash A$ does not because in $H[Q]$, A is defined only by tautological clauses.

    On the other hand, if A has no refutation in P, then take $D = \{A\}$ and choose $\mathtt{T}$ as before to ensure that $H[P] = P \; isa \; D \; isa \; \mathtt{T}$ and $H[Q] = Q \; isa \; D \; isa \; \mathtt{T}$ are both well-formed. Then $H[Q] \vdash A$ has an isa-proof while $H[P] \vdash A$ does not because the clauses that define p in P override the unit clause A in D.

Now consider the case when $\nabla_P = \nabla_Q$ and $F(P) \neq F(Q)$. Again, assume without loss of generality that there exists a cs-clause

$$\mathtt{C}_P : \ A \leftarrow s_P \ \square \ \mathbf{B}_P \in F(P) \setminus F(Q).$$

There is no loss of generality in further assuming that there exists no $\mathtt{C}_Q \in F(Q)$ such that $\mathtt{C}_Q$ subsumes $\mathtt{C}_P$ because, if it were so, then we could reason on $\mathtt{C}_Q \in F(Q) \setminus F(P)$ relying on the additional assumption. In fact, if it existed $\mathtt{C}'_P$ that subsumes $\mathtt{C}_Q$, then $\mathtt{C}'_P$ would also subsume $\mathtt{C}_P$ and this would contradict the hypothesis that $F(P)$ is in weak canonical form.

Now partition the atoms of $\mathbf{B}_P$ according to their annotation as follows:

$$
\begin{aligned}
S &= \{p(\mathbf{t}) \mid p(\mathbf{t}) \in \mathbf{B}_P \text{ and } p \in \text{STAT}(P) \setminus \mathit{Defn}(P)\}, \\
D &= \{p(\mathbf{t}) \mid p(\mathbf{t}) \in \mathbf{B}_P \text{ and } p \in \text{DYN}(P)\}, \\
E &= \{p(\mathbf{t}) \mid p(\mathbf{t}) \in \mathbf{B}_P \text{ and } p \in \text{EXT}(P)\}.
\end{aligned}
$$

That $\{S, D, E\}$ is a partition on $\mathbf{B}_P$ follows from the definition of D_P because, being $\mathtt{C}_P$ a cs-clause of $D_P \uparrow \omega$, $\mathit{Pred}(\mathbf{B}_P) \subseteq \mathit{Pub}(P)$. We define also a fourth set of cs-clauses as

$$R = \{p(\mathbf{t}) \leftarrow p(\mathbf{t}) \mid \text{ there exists } A \leftarrow s_Q \ \square \ \mathbf{B}_P \in F(Q) \ \text{ such that } \ p \in s_Q \setminus s_P\}.$$

In other words, R contains tautological clauses for all (and only) the predicates p such that (i) p does not belong to the context s_P of the cs-clause $\mathtt{C}_P$ of $F(P)$ and, (ii) p belongs to the context s_Q of any of the cs-clauses of $F(Q)$ that has the same head and the same body as $\mathtt{C}_P$.

Finally, we define the set of tautologies:

$$T = \{p(\mathbf{t}) \leftarrow p(\mathbf{t}) \mid p \in (\text{STAT}(P) \setminus \mathit{Defn}(P)) \cup (\text{STAT}(Q) \setminus \mathit{Defn}(Q)) \text{ and } p \notin \mathit{Pred}(\mathbf{B}_P)\}$$

Thus T contains tautological clauses for the static predicates of P and Q which are not defined in P or in Q and do not occur in $\mathbf{B}_P$.

Now, let $H[P]$ and $H[Q]$ be the hierarchies defined as follows:

$$
\begin{aligned}
H[P] &= DR \text{ isa } P \text{ isa } STE, \\
H[Q] &= DR \text{ isa } Q \text{ isa } STE
\end{aligned}
$$

where DR and STE are, respectively, the differential programs $D \cup R$ and $S \cup T \cup E$. Being $\nabla_P = \nabla_Q$, both $H[P]$ and $H[Q]$ are well-formed. We will prove that $H[P] \not\equiv_{isa} H[Q]$ by showing that A, the head of $\mathtt{C}_P$, is such that $A \in M_{isa}(H[P])$ whereas $A \notin M_{isa}(H[Q])$.

We first show that $A \in M_{isa}(H[P])$ showing that $A \in hd(F(DR \lhd (P \lhd STE)))$. We start computing $F(P \lhd STE)$. It is easy to see that $F(P \lhd STE) = F(F_l(P) \cup SE)$ (where SE denotes $S \cup E$) because the choice of STE guarantees that $F_r(STE) = F(STE) = SE$. Then, letting $\mathbf{D}$ denote the subset of dynamic atoms of $\mathbf{B}_P$, clearly

$$A \leftarrow s_P \ \square \ \mathbf{D} \in F(P \lhd STE)$$

because $A \leftarrow s_P \ \square \ \mathbf{B}_P \in F_l(P)$ and we can unfold every atom in $\mathbf{B}_P \setminus \mathbf{D}$ with the unit clauses of SE. Now, by definition, $F(DR \lhd (P \lhd STE)) = F(F_l(DR) \cup F_r(P \lhd STE))$ and we can show that $A \leftarrow s_P \ \square \ \mathbf{D} \in F_r(P \lhd STE)$.

To prove this we need to show that $\mathit{Pred}(A) \not\subseteq \nabla_{DR}$ as well as that $s_P \cap \nabla_{DR} = \emptyset$. First note that $\mathit{Pred}(A) \not\subseteq \nabla_D$ because (i) $\mathit{Pred}(A) \not\subseteq \mathit{Pred}(\mathbf{B}_P) \cap \text{DYN}(P)$ being $A \leftarrow s_P \ \square \ \mathbf{B}_P \in$

24

$F(P)$, and (ii) $Pred(\mathbf{B}_P) \cap \mathrm{DYN}(P) = Pred(D) = \nabla_D$ by construction of D. Furthermore, by construction $Pred(R) = \nabla_R$, and the choice of R guarantees that $Pred(A) \not\subseteq Pred(R)$ because none of the cs-clauses in $F(Q)$ whose head's predicate symbol is in $Pred(A)$ have the predicate of A in their context (owing to the δ_P-abstraction on $D_Q \uparrow \omega$). Hence we can conclude that $Pred(A) \not\subseteq \nabla_{DR}$ because $\nabla_{DR} = \nabla_D \cup \nabla_R$.

The proof that $s_P \cap \nabla_{DR} = \emptyset$ follows the same idea. In fact, by construction, none of the predicates of s_P appears in R and, consequently, $s_P \cap Pred(R) = s_P \cap \nabla_R = \emptyset$. Finally, $s_P \cap Pred(D) = \emptyset$ (whence $s_P \cap \nabla_D = \emptyset$) because otherwise $s_P \cap Pred(\mathbf{B}_P) \neq \emptyset$ and $A \leftarrow s_P \,\square\, \mathbf{B}_P$ would not be part of $F(P)$ (owing again to the δ-abstraction on $D_P \uparrow \omega$).

Now, $A \leftarrow s_P \,\square\, \mathbf{D} \in F_r(P \lhd STE)$ implies that $A \leftarrow s_P \cup Pred(\mathbf{D}) \,\square\, \in F(DR \lhd (P \lhd STE))$. In fact, $F(DR \lhd (P \lhd STE)) = F(DR \cup F_r(P \lhd STE))$ because the choice of DR guarantees that $F_l(DR) = F(DR) = DR$. Hence, $A \leftarrow s_P \cup Pred(\mathbf{D}) \,\square\,$ can be obtained from $A \leftarrow s_P \,\square\, \mathbf{D} \in F_r(P \lhd STE)$ by unfolding the atoms of $\mathbf{D}$ with the unit clauses of DR. Finally, $A \in M_{isa}(H[P])$ because:

$$M_{isa}(H[P]) \quad = \quad hd(F(DR \lhd (P \lhd STE))).$$

We conclude the proof showing that $A \notin M_{isa}(H[Q])$. First observe that $A \notin DR \cup STE$. To see this note that $A \notin Pred(\mathbf{B}_P)$ because otherwise $A \leftarrow s_P \,\square\, \mathbf{B}_P$ would not be in $F(P)$ (owing to the δ-abstraction on $D_P \uparrow \omega$). But then $A \notin D \cup S \cup E$ because these three sets partition $\mathbf{B}_P$, and $A \notin DR \cup STE$ follows because R and T consist only of tautological clauses.

Now by definition, $A \in M_{isa}(H[Q])$ if and only if $A \in hd(F(DR \lhd (Q \lhd STE)))$ if and only if $A \in hd(F(DR) \prec (F(Q) \prec (STE)))$. Now, since $A \notin DR \cup STE$, $A \in M_{isa}(H[Q])$ holds only if there exists a cs-clause $\mathtt{C}_Q \in F(Q)$ such that

$$\mathtt{C}_Q = A \leftarrow s_Q \,\square\, \mathbf{B}_Q \in \mathcal{F}(Q)$$

and both the following two conditions hold:

(i) $\mathbf{B}_Q \subseteq D \cup S \cup E$;

(ii) there exists $\mathtt{C}'_Q \in F(Q) \prec F(STE)$ such that $\mathtt{C}'_Q$ is obtained from $\mathtt{C}_Q$ by unfolding the static and extensible predicates of $\mathbf{B}_Q$ with the unit clauses in STE, and $\mathtt{C}'_Q \in F_r(F(Q) \prec F(STE))$

Now, by our initial assumption, if $\mathtt{C}_Q \in F(Q)$, then $\mathtt{C}_Q$ does not subsumes $\mathtt{C}_P$. Therefore, either there exists $B \in \mathbf{B}_Q \setminus \mathbf{B}_P$ or there exists $p \in s_Q \setminus s_P$. Consider the two cases separately.

If $B \in \mathbf{B}_Q \setminus \mathbf{B}_P$, then $B \notin D \cup S \cup E$ because $B \notin \mathbf{B}_P$ and $D \cup S \cup E = \mathbf{B}_P$.

If $p \in s_Q \setminus s_P$, let $\mathtt{C}'_Q = A \leftarrow s'_Q \,\square\, \mathbf{D}_Q \in F(Q) \prec F(STE)$ be the cs-clause obtained from $\mathtt{C}_Q$ by unfolding. Clearly, $s'_Q = s_Q$, since no dynamic predicate is defined in STE, and hence $p \in s'_Q \setminus s_P$. Furthermore, by construction, $p(\mathbf{t}) \leftarrow p(\mathbf{t}) \in R$: hence $s'_Q \cap Defn(DR) \neq \emptyset$ and this, in turn implies that $\mathtt{C}'_Q \notin F_r(F(Q) \prec F(STE))$.

Since the choice of $\mathtt{C}_Q$ was arbitrary, the previous argument applies to all the cs-clauses of $F(Q)$ that have A as their head. But this implies that $A \notin F(DR) \prec (F(Q) \prec F(STE))$ and hence, by definition, $A \notin M_{isa}(H[Q])$. Since we showed that $A \in M_{isa}(H[P])$, this together with Proposition 3.13 completes the proof. ∎

5 Concluding Remarks

There are two aspects that are central to the programming paradigm we have proposed. On one side we contend that, owing to the different forms of interaction among the component modules, the composition of differential programs in *isa* hierarchies provides a powerful and quite effective tool for modular programming. On the other side, in spite of its complexity, we have shown that the logical foundations of this operator can be traced back to the theory of logic programming and that the computational properties of the *isa* composition can be characterized mathematically in terms of a compositional and fully abstract semantics.

It is clear, however, that other features are needed to make this programming paradigm effective as a support for programming-in-the-large. In this section we address some of these features and we show how they can be accounted for in our framework.

We first lift the assumption on the linearity of the *isa* hierarchies by allowing the components of a program to be organized into an *isa* schema that may be either a tree or a DAG. In this case, querying a module of schema results into a query for the hierarchy that has that module as its leaf. In the editor program of Section 2, for instance, the *isa* schema could be structured as follows:

```
                    EDITOR
                      ↑
                    EMAX
                ↗          ↖
      RCS-EMAX           LaTeX-EMAX
```
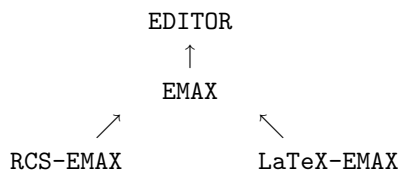
A query for `EMAX` would then correspond to a query for the hierarchy `EMAX` *isa* `EDITOR` whereas a query for, say, `RCS-EMAX` would result into a query for `RCS-EMAX` *isa* `EMAX` *isa* `EDITOR`.
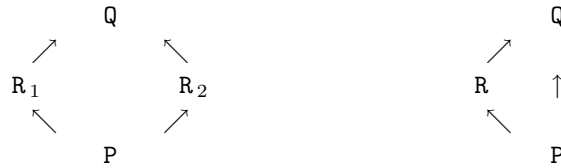
When the schema is a tree, as in the above diagram, every module is connected to its ancestors by a linear hierarchy. Hence the result of evaluating a query, as well as the computational properties of the component modules are subject to the semantics we have discussed in Sections 3 and 4. When the schema is a DAG, instead, the hierarchy that connects a module with its ancestors can be itself a DAG and, consequently, the behavior of a predicate call in that module may be subject to multiple inheritance.

5.1 Multiple Inheritance

Multiple inheritance has long been a controversial feature in object oriented systems. The reason of this controversy is that, while multiple inheritance is potentially very effective as a tool for code sharing and code reuse, it is difficult, in general, to give a semantic characterization to the behavior that arises from the inheritance of conflicting definitions. In a logic programming framework, instead, the existence of conflicting definitions represents a far less serious concern because, as argued by [20], a predicate (a relation) may reasonably be defined by cases, each case in a different ancestor of the same module.

This observation applies naturally to our case: when the *isa* schema is a DAG, predicates are inherited along the paths of the DAG in ways similar to those outlined in the previous sections. There are however the classical problems related to the semantics of multiple inheritance that carry over in our framework too, and must be dealt with.

We can exemplify these problems looking at the following two diagrams.

$$
\begin{array}{ccccccc}
 & & \text{Q} & & & & \text{Q} \\
 & \nearrow & & \nwarrow & & \nearrow & \\
\text{R}_1 & & & \text{R}_2 & \text{R} & & \uparrow \\
 & \nwarrow & & \nearrow & & \nwarrow & \\
 & & \text{P} & & & & \text{P}
\end{array}
$$

Consider first the diagram on the left and assume that Q defines a predicate, say p, that is not defined by any of the remaining components of the DAG. The natural question is: should P inherit the definition of p from Q once, or twice along the two paths that connect P with Q? In fact the two choices are equivalent in our framework: the only difference between the two interpretations is that in the latter the answers for p get duplicated. However, duplication is not an issue in our proof and model theories because we have defined the semantics of a program as the *set* of atoms that can be proved in that program.

More critical problems arise from the interplay between multiple inheritance and overriding. Consider, in the left diagram, the case when Q and R_2 both define an overridable predicate, say p, that is defined neither by R_1 nor by P. Here the question is whether P should inherit the definition coming from Q, along the path P *isa* R_1 *isa* Q, or whether it should not because R_2 defines p and R_2 is closer to P than Q. The diagram on the right poses a similar problem: assume for instance that both Q and R define an overridable predicate, p again, that is not defined by P. Now the question is: should Q be considered "closer" to P than R or viceversa? In other words, should P inherit the definition of p from R or Q?

The easiest solution to these problems is, of course, to accept the inheritance of p from Q to P in both of the above cases. This choice corresponds to consider the two DAGs equivalent, respectively, to the trees obtained by duplicating P and letting each copy have a single parent. Alternatively, as it is done in several object oriented systems currently in use, we could resort to algorithms that impose a total order over the elements of the DAG by extending the order that is expressed by the DAG itself. In either case, we would be able to express the semantics of each component of the DAG in terms of the semantics of that module within each of the linear hierarchies it belongs to.

We should mention, however, that some of the recent proposals of object oriented languages in the literature ([12] for instance) adopt a more elegant solution by resorting to the following, quite general, overriding policy:

> a module (class) P inherits a predicate (method) from an ancestor Q if and only if that predicate is not defined by any intervening module R such that P *isa* R and R *isa* Q.

According to this view, P would not inherit the definition of p in either of the above two diagrams: in both cases, in fact, an intervening module that defines p does actually exist. Adopting this approach in our framework would be possible but would require substantial modifications in the definition of the operator $\ll$ of semantic composition: in fact, the overriding policy underlying the approach creates inter-dependencies among paths that, instead, are independent in the DAG.

27

5.2 Encapsulation and Information Hiding

Encapsulation is a major concept in the suite of notions comprising the paradigm of programming-in-the large: it is an essential software-engineering technique which helps define modules with interfaces that restrict the access to the predicates (methods) of a module only through the *public* methods of the module's interface.

According to this informal definition, the use of internal predicates does capture a form of encapsulation by allowing a module to hide part of its predicates and to export only the public predicates it defines. However, more sophisticated forms of hiding are clearly desirable. For instance we may want to share the visibility of a predicate among the components of a hierarchy, but hide that predicate from users of the hierarchy. Mechanisms like these are discussed in [26] and [10] and turn out to be useful in several applications. Consider for instance the case of a schema like the following:
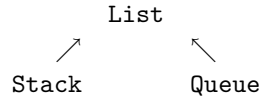
$$\text{List}$$
$$\nearrow \qquad \nwarrow$$
$$\text{Stack} \qquad \text{Queue}$$

This schema could be used, for instance, to implement two *abstract modules* `Stack` and `Queue` that use `List` as the same implementation module. `List` would define all the predicates, like *append*, implementing the primitive operations on lists: these primitives, in their turn, should be visible from `Stack` and `Queue` but should be hidden from users of the two abstract modules. To model this behavior, we would need mechanisms that allow *append* to be shared between `List` and `Stack` (or `Queue`) and to be hidden from users of the hierarchy `Stack` *isa* `List` (`Queue` *isa* `List`). As it turns out, we can simulate these mechanisms quite naturally in our framework, by defining a new operator on differential programs as outlined below.

First, given an annotated universe A and a set of predicate symbols π, denote with A_π the new universe whose annotation coincides with the annotation of A on all the predicate symbols except those of π that are treated as internal predicates in A_π. Then, for every program P over a universe A, and every set of predicate symbols π, denote with $\pi \bullet P$ the new program obtained by interpreting P in the universe A_π. It is immediate to see how we can define the model theoretic semantics of $\pi \bullet P$: simply, take $M(P)$ and filter out all the atoms whose predicate symbols are internal in A_π. Furthermore, with a corresponding projection, we can clearly compute $[\![\pi \bullet P]\!]$ from $[\![P]\!]$.

Now we can define a corresponding operator for hierarchies of differential programs as follows. Given an *isa* hierarchy H of programs over the annotated universe A, denote with $\pi \bullet H$ the new hierarchy whose model theoretic semantics is defined as follows:

$$M_{isa}(\pi \bullet H) = \pi \bullet M_{isa}(H)$$

Although somewhat loose because, strictly speaking $M_{isa}(H)$ is not a differential program, the notation should be self-explanatory: to compute $M_{isa}(\pi \bullet H)$, we first compute the semantics of H according of the annotation of A and the we filter out all the atoms relative to the predicates that occur in π.

Given this interpretation, we can model the expected behavior for our example by means of the following construction:

$$\{append, \cdots\} \bullet (\text{Stack } isa \text{ List}).$$

We content ourselves with this intuitive picture as further investigations are beyond the scope of the present paper. Instead, what we would like to emphasize here is that taking the *isa*

composition as a basic operator, we can then define several other powerful structuring mechanisms for the development of large programs. Notably, we can account for most of the features proposed in the approaches that are briefly surveyed in the rest of this section.

5.3   Related work

We can classify related works according to the three following main streams.

Program Composition

Several proposals in the literature follow an approach to modular programming based on program composition. We already mentioned the "open programs" introduced by Bossi et al. in [7], the modular frameworks of Mancarella and Pedreschi [19], Gaifman and Shapiro [13], and of Brogi, Lamma and Mello in [9].

The novelty of our proposal is in the type of composition mechanisms we have considered as well as in the domain chosen for the semantic characterization. Our composition operator provides a uniform semantics for the composition mechanisms proposed in the existing approaches and extends them with an explicit treatment of overriding. The use of *internal* predicates, and its extensions outlined in the previous subsection, provide also a formal account of information hiding richer than those proposed by Gaifman and Shapiro in [13] and comparable to those proposed in a recent paper by Brogi et al. [10]. This last paper has motivations similar to ours. However, the advantage of our framework is that with a single operator, *isa*, we can account for an even wider range of applications than the operators defined in [10]. Notably, the framework of [10] has no account of overriding as one of the possible features and it appears difficult to extend that set of operators to account for it semantically. Furthermore, our compositional semantics, although complex, is more abstract than the T_P operator used in [10] and hence, more practical as a basis for defining semantic-based tools for program development.

We should finally mention the paper [16] by Laesen and Vermeir: it presents a fixpoint semantics for a composition operator for *Ordered theories* which has several analogies with our *isa* operator. However, their approach is hardly comparable with ours in terms of motivations: they use program composition to model powerful forms of reasoning about inheritance hierarchies in artificial intelligence [29] and, as such, they are not interested in any of the issues, notably compositionality, that we have addressed in the present paper.

Modular Languages

An orthogonal approach to modular programming was instead motivated by the work on embedded implications by D. Miller in [22]. Embedded implications as structuring tools have been used by a number of other authors in the attempt to model adequate scope rules for modular and object-oriented programming in logic programming. Some of these proposals are outline below.

In [23] Monteiro and Porto proposed Contextual Logic Programming as a modular logic programming language based on a new type of implication goal, called *extension goal*. The corresponding connective, the *context extension* operator ≫, models a rule of lexical scope that has essentially the same semantic connotation as *static* inheritability (with overriding) in our framework.

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 but their approach to the semantics is solely transformational. 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. Again, the difference is that their semantics framework applies to *complete* hierarchies, like our model theoretic semantics, but there is no attempt to give any formal account of compositionality.

A compositional semantics of inheritance with overriding is proposed in [11], 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 inheritability. In contrast to that case, the choice of context-sensitive interpretations, allows us to have a uniform treatment of the two mechanisms.

In [14], Giordano *et al.* present an elegant model for static and dynamic inheritability in a modular language that combines classical and intuitionistic (embedded) implications. In a related paper [4], Badaloni *et al.* present an new language where modules are defined directly (i.e. without resorting to embedded imlications) by means of modal operators associated with the modules of a program. The relations with our approach are somewhat loose in that (i) neither of these two papers gives a formal account of overriding and (ii) the model-theoretic semantics of the two languages is given in terms of a possible-worlds semantics based on Kripke interpretations.

A modular extension to logic programming was also proposed by Sannella and Wallen in [27], 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.

Deductive OO Languages

In the context of deductive object oriented systems, there have been several attempts at combining inheritance with deductive programming languages within clean mathematical settings.

In *L&O* [20], the semantics of inheritance and overriding is given only indirectly by translating *L&O* program to logic programs and, hence, it provides little insight into the relationships between inheritance, overriding and deduction.

Languages like LOGIN [1], LIFE [2], F-Logic [15] and GULOG [12] represent further proposals in this area. However, in these approaches the resulting languages are based on a complex-object data model where objects are represented by (generalizations of) first order terms, rather than by sets of clauses as in our case. Hence, although the functionalities of inheritance are comparable to ours, these proposals differ substantially from ours both in terms of motivations and technicals solutions.

Acknowledgments

We would like to thank the anonymous referees of a previous version of this paper for their insightful comments and very helpful suggestions.

References

[1] H. Aït-Kaci and R. Nasr. Login: a Logic Programming Language with built-in Inheritance. *Journal of Logic Programming*, 3:182–215, 1986.

[2] H. Aït-Kaci and A. Podelski. Towards a Meaning of LIFE. Technical Report 11, Digital Paris Research Labs, 1991.

[3] 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.

[4] M. Baldoni, Giordano, and A. Martelli. A Multimodal Logic to Define Modules in Logic Programming. In D. Miller, editor, *Proceedings of the International Logic Programming Symposium ILPS'93*, pages 473–487. The MIT Press, 1993.

[5] A. Bossi, M. Bugliesi, M. Gabbrielli, G. Levi, and M.C. Meo. Differential Logic Programming. In *Proceedings ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '93*, pages 359–370. ACM, 1993.

[6] A. Bossi, M. Bugliesi, M. Gabbrielli, G. Levi, and M.C. Meo. Differential Logic Programming. Technical Report CS-R9363, CWI, Amsterdam, NL, 1993.

[7] A. Bossi, M. Gabbrielli, G. Levi, and M. C. Meo. A Compositional Semantics for Logic Programs. *Theoretical Computer Science*, 122(1-2):3–47, 1994.

[8] 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.

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

[10] A. Brogi, P. Mancarella, D. Pedreschi, and F. Turini. Modular Logic Programming. *ACM Transactions on Programming Languages and Systems*, 16(4):1131–1398, 1994.

[11] 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.

[12] G. Dobbie and R. Topor. A Model for Inheritance and Overriding in Deductive Object-Oriented Systems. In *Sixteen Australian Computer Science Conference*, January 1993.

[13] 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.

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

[15] M. Kifer, G. Lausen, and J. Wu. Logical Foundations for Object-Oriented and Frame-Based Languages. Technical Report TR-93/06, Department of Computer Science, SUNY at Stony Brook, 1993. (accepted to Journal of ACM).

[16] E. Laesen and D. Vermeir. A Fixpoint Semantics for Ordered Logic. *Journal of Logic and Computation*, 1(2):159–185, 1990.

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

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

[19] 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.

[20] F.G. McCabe. *Logic and Objects.* Prentice Hall International, London, 1992.

[21] P. Mello, A. Natali, and C. Ruggieri. Logic programming in a software engineering perspective. In L. Lusk and R.A. Overbeek, editors, *Proc. 1989 North American Conf. on Logic Programming*, pages 451–458. The MIT Press, 1989.

[22] D. Miller. A logical analysis of modules in logic programming. *Journal of Logic Programming*, 6:79–108, 1989.

[23] L. Monteiro and A. Porto. Contextual logic programming. In G. Levi and M. Martelli, editors, *Proc. 6th Int. Conference on Logic Programming*, pages 284–302. 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. 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.

[26] 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.

[27] 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.

[28] 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.

[29] D. S. Touretzky. *The Mathematics of Inheritance Systems.* Morgan Kaufmann, Los Altos, CA, 1986.

A Appendix

We prove the lemmas needed in the proof of Theorem 4.11.

Lemma A.1 Let P and Q be differential programs such that P isa Q is a well-formed hierarchy and let $P \lhd Q$ be the differential program $P \cup \phi(Q^\star)$, where $\phi(Q^\star)$ is defined according to Definition 3.3. Then

$$D_{\phi(Q^\star)} \uparrow \omega \;\; = \;\; \phi(D_Q \uparrow \omega) \setminus S$$

where $S = \{A \leftarrow s \,\square\, \mathbf{B} \in \mathcal{C} \mid s \cap Defn\,(P) \neq \emptyset \text{ or } Pred(A) \subseteq \text{DYN}(Q) \cap Defn\,(P)\}$.

PROOF. First observe that the hypothesis that P isa Q is well-formed implies that every static predicate of Q is also defined in Q: hence, by definition, $\text{STAT}(Q) \cap Open(Q) = \emptyset$. But then, since ϕ renames only static predicates in Q, and none of these is open, we have that $Open(Q) = Open(\phi(Q))$. This, in turn, implies that

$$\phi(D_Q(I)) \;\; = \;\; D_{\phi(Q)}(I)$$

for every cs-interpretation I. Owing to this identity, we can prove the claim showing that:

$$D_{\phi(Q^\star)} \uparrow n \;\; = \;\; D_{\phi(Q)} \uparrow n \setminus S.$$

We prove the two inclusions separately.

($\subseteq$) First note that $Open(\phi(Q^\star)) \subseteq Open(\phi(Q))$ because the subsetting $\star$ deletes only clauses defining dynamic predicates. Furthermore, since (clearly) $\phi(Q^\star) \subseteq \phi(Q)$, it follows that for every cs-clause $\mathsf{c} \in D_{\phi(Q^\star)} \uparrow n$, $\mathsf{c} \in D_{\phi(Q)} \uparrow n$.

To complete the proof we need to show that $\mathsf{c} \notin S$, but this is easy to verify by induction on n. For $n = 0$ the proof is trivial. For $n > 0$ we can reason as follows: if c is the cs-clause $A \leftarrow s \,\square\, \mathbf{L}_1, \ldots \mathbf{L}_k$, then c is obtained from a clause $A \leftarrow B_1, \ldots B_k \in \phi(Q^\star)$, and k cs-clauses in $D_{\phi(Q^\star)} \uparrow (n-1)$ which, by the inductive hypothesis are not contained in S. Now that $\mathsf{c} \notin S$ follows immediately from the definition of $D_{\phi(Q^\star)}$.

($\supseteq$) The proof of this inclusion follows by induction on $n \geq 0$. The base case, when $n = 0$ is trivial since both the sets are empty. For the inductive case ($n > 0$), take $\mathsf{c} \in D_{\phi(Q)} \uparrow n \setminus S$ and let c be the cs-clause $\mathsf{c} = \; A \leftarrow s_0 \cup s_1 \ldots \cup s_k \,\square\, \mathbf{L}_1, \ldots, \mathbf{L}_k$. Then, by definition of $D_{\phi(Q)}$,

> there exists $A \leftarrow B_1, \ldots, B_k \in Ground(\phi(Q))$,
> for every $i \in [1 \ldots k]$ there exists $\mathsf{c}_i = B_i \leftarrow s_i \,\square\, \mathbf{L}_i \in D_{\phi(Q)} \uparrow (n-1) \cup \mathtt{Id}_{Open(\phi(Q))}$,
> $s_0 = \{Pred(B_i) \mid Pred(B_i) \subseteq \text{DYN}(\phi(Q)) \text{ and } \mathsf{c}_i \notin \mathtt{Id}_{Open(\phi(Q))}\}$.

Being $Q^\star \subseteq Q$, clearly $\text{DYN}(\phi(Q^\star)) \subseteq \text{DYN}(\phi(Q))$ and, since ϕ does not rename any dynamic predicate, $\text{DYN}(\phi(Q)) = \text{DYN}(Q)$. Now, since $\mathsf{c} \notin S$, we have $Pred(A) \not\subseteq \text{DYN}(\phi(Q^\star)) \cap Defn\,(P)$ and, consequently,

(i) $A \leftarrow B_1, \ldots, B_k \in Ground(\phi(Q^\star))$.

Now consider the c_is in the set $D_{\phi(Q)} \uparrow (n-1) \cup \mathtt{Id}_{Open(\phi(Q))}$ and distinguish two cases:

(ii) $\mathsf{c}_i \in D_{\phi(Q)} \uparrow (n-1) \setminus \mathtt{Id}_{Open(\phi(Q))}$: from $\mathsf{c} \notin S$ it follows that $(s_0 \cup s_1 \ldots \cup s_k) \cap Defn\,(P) = \emptyset$ and hence, $Pred(B_i) \not\subseteq \text{DYN}(\phi(Q)) \cap Defn\,(P)$ and $s_i \cap Defn\,(P) = \emptyset$. Therefore $\mathsf{c}_i \notin S$ and hence, by the inductive hypothesis $\mathsf{c}_i \in D_{\phi(Q^\star)} \uparrow (n-1)$. Moreover, since $\mathtt{Id}_{Open(\phi(Q^\star))} \subseteq \mathtt{Id}_{Open(\phi(Q))}$, $\mathsf{c}_i \notin \mathtt{Id}_{Open(\phi(Q^\star))}$.

(iii) $c_i \in \mathrm{Id}_{\phi(Q)}$: then clearly $Pred(B_i) \subseteq Open(\phi(Q))$, and since, by (i) above, $Pred(B_i) \subseteq Pred(\phi(Q^\star))$ we have also that $c_i \in \mathrm{Id}_{\phi(Q^\star)}$.

Now $c \in D_{\phi(Q^\star)} \uparrow n$ follows from (i), (ii) and (iii) and the definition of $D_{\phi(Q^\star)}$. ∎

Lemma 4.9 Let P and Q be differential programs such that P isa Q is a well-formed hierarchy. Then

$$F(\phi(Q^\star)) = F_r(Q)$$

where $\phi(Q^\star)$ and $F_r(Q)$ are defined according to Definitions 3.3 and 4.8 respectively.

PROOF. Let us introduce two cs-interpretations R and S as follows

$$
\begin{aligned}
R &= \{A \leftarrow s \,\square\, \mathbf{L} \in \mathcal{C} \mid Pred(A) \subseteq \mathrm{STAT}(Q) \cap Defn(P)\} \\
S &= \{A \leftarrow s \,\square\, \mathbf{L} \in \mathcal{C} \mid s \cap Defn(P) \neq \emptyset \text{ or } Pred(A) \subseteq \mathrm{DYN}(Q) \cap Defn(P)\}.
\end{aligned}
$$

By definition, $F(\phi(Q^\star)) = \alpha(D_{\phi(Q^\star)} \uparrow \omega)$ and it is easy to see that $F_r(Q) = \alpha(D_Q \uparrow \omega) \backslash (R \cup S)$. Then, to prove the claim, we have to show that:

$$\alpha(D_{\phi(Q^\star)} \uparrow \omega) \;=\; \alpha(D_Q \uparrow \omega) \setminus (R \cup S). \tag{1}$$

By Lemma A.1, we already have that:

$$\alpha(D_{\phi(Q^\star)} \uparrow \omega) = \alpha(\phi(D_Q \uparrow \omega) \setminus S). \tag{2}$$

Now we can reason on the properties of the abstractions ι, δ and wcf that are composed in α.

Consider ι first. Since ι is a subsetting, for every cs-interpretation I, $\iota(I \setminus S) = \iota(I) \setminus S$ and clearly $\iota(I \setminus (R \cup S)) = \iota(I \setminus R) \setminus S = \iota(I) \setminus (R \cup S)$.

Now, since P isa Q is well-formed by hypothesis, $\mathrm{STAT}(Q) \cap Open(Q) = \emptyset$. Hence, none of the static predicates of Q occurs in the body of a cs-clause of $D_Q \uparrow \omega$. From that, and the fact that ϕ renames only the predicates in $\mathrm{STAT}(Q) \cap Defn(P)$, we clearly have that $\iota(\phi(D_Q \uparrow \omega) = \iota(D_Q \uparrow \omega \setminus R)$. Therefore, $\iota(\phi(D_Q \uparrow \omega) \setminus S) = \iota(\phi(D_Q \uparrow \omega)) \setminus S = \iota(D_Q \uparrow \omega \setminus R) \setminus S$ and hence:

$$\iota(\phi(D_Q \uparrow \omega) \setminus S) = \iota(D_Q \uparrow \omega) \setminus (R \cup S). \tag{3}$$

Now consider δ: we prove that for every cs-interpretation I, the following holds:

$$\delta(I \setminus (R \cup S)) \;=\; \delta(I) \setminus (R \cup S). \tag{4}$$

($\subseteq$) Let $c = A \leftarrow s \,\square\, \mathbf{B} \in \delta(I \setminus (R \cup S))$ and $c' = A \leftarrow s' \,\square\, \mathbf{B}$ the cs-clause in $I \setminus (R \cup S)$ which exists, by Definition 4.6. Clearly $c \in \delta(I)$ and we have only to show that $c \notin (R \cup S)$. By contradiction, assume $c \in (R \cup S)$. Now, if $c \in (R \cup S)$ because $Pred(A) \subseteq \mathrm{STAT}(Q) \cap Defn(P)$, or $Pred(A) \subseteq \mathrm{DYN}(Q) \cap Defn(P)$, then $c' \in (R \cup S)$ for the same reasons, and this contradicts the initial assumption on c'. On the other hand, if $c \in (R \cup S)$ because $s \cap Defn(P) \neq \emptyset$, then again $c' \in (R \cup S)$ since $s \subseteq s'$ and then $s' \cap Defn(P) \neq \emptyset$.

($\supseteq$) Let $c = A \leftarrow s \,\square\, \mathbf{B} \in \delta(I) \setminus (R \cup S)$ and $c' = A \leftarrow s' \,\square\, \mathbf{B}$ the cs-clause in I which, again, exists by Definition 4.6. By contradiction, assume $c' \in (R \cup S)$ and consider the three possibilities as in the previous case. In the first two cases, we can reason exactly as before to conclude that $c \in (R \cup S)$, which is again a contradiction. As for the third case, there are two more possibilities: $s' = s$ or $s' = s \cup Pred(A)$. If $s' = s$, again $c = c' \in (R \cup S)$; if $s' = s \cup Pred(A)$, $Pred(A) \subseteq \mathrm{DYN}(Q)$ but since we have already excluded that $Pred(A) \subseteq$

DYN$(Q) \cap Defn(P)$, then $s \cap Defn(P) = s' \cap Defn(P)$. Hence, $s \cap Defn(P) \neq \emptyset$ and then $\mathsf{c} \in (R \cup S)$.

Finally, consider wcf: we show that, for every cs-interpretation I,

$$wcf(I \setminus (R \cup S)) \quad = \quad wcf(I) \setminus (R \cup S). \tag{5}$$

($\subseteq$) Take $\mathsf{c} \in wcf(I \setminus (R \cup S))$ and assume by contradiction that $\mathsf{c} \notin wcf(I) \setminus (R \cup S)$. Then either $\mathsf{c} \in (R \cup S)$ or there exists $\mathsf{c}' \in I$ such that c' subsumes c strictly.

We can readily exclude the first case since $\mathsf{c} \in wcf(I \setminus (R \cup S))$ implies that $\mathsf{c} \in I \setminus (R \cup S)$ and hence $\mathsf{c} \notin (R \cup S)$ which is a contradiction. For the second case, first note that since $\mathsf{c} \in wcf(I \setminus (R \cup S))$, and since c' subsumes c, we must have that $\mathsf{c}' \in (R \cup S)$. Now we can show that if $\mathsf{c}' \in (R \cup S)$ and c' subsumes c strictly, then $\mathsf{c} \in (R \cup S)$ (which is a contradiction). To see this we reason as for the δ abstraction: since c' subsumes c strictly c and c' must be of the form $\mathsf{c} = A \leftarrow s \square \mathbf{B}$ and $\mathsf{c}' = A \leftarrow s' \square \mathbf{B}'$ where $\mathbf{B}' \subseteq \mathbf{B}$ and $s' \subseteq s$.

Now, if $\mathsf{c}' \in R \cup S$ because $Pred(A) \subseteq$ STAT$(Q) \cap Defn(P)$ or $Pred(A) \subseteq$ DYN$(Q) \cap Defn(P)$, then $\mathsf{c} \in R \cup S$ for the exact same reasons. Otherwise, $\mathsf{c}' \in R \cup S$ because $s' \cap Defn(P) \neq \emptyset$ but then $\mathsf{c} \in R \cup S$ because $s' \subseteq s$ implies that $s \cap Defn(P) \neq \emptyset$.

($\supseteq$) This is obvious since $\mathsf{c} \in wcf(I) \setminus (R \cup S)$ implies that $\mathsf{c} \in I \setminus (R \cup S)$ and that for every $\mathsf{c}' \in I \setminus (R \cup S)$, c subsumes c' strictly. But then $\mathsf{c} \in wcf(I \setminus (R \cup S))$ by definition.

Now we conclude the proof of (1) as follows:

$$
\begin{aligned}
\alpha(D_{\phi(Q^\star)} \uparrow \omega) &= &&\text{by (2)} \\
\alpha(\phi(D_Q \uparrow \omega) \setminus S) &= &&\text{by Definition 4.6} \\
wcf(\delta(\iota(\phi(D_Q \uparrow \omega) \setminus S))) &= &&\text{by (3)} \\
wcf(\delta(\iota(D_Q \uparrow \omega) \setminus (R \cup S))) &= &&\text{by (4)} \\
wcf(\delta(\iota(D_Q \uparrow \omega)) \setminus (R \cup S)) &= &&\text{by (5)} \\
wcf(\delta(\iota(D_Q \uparrow \omega))) \setminus (R \cup S) &= &&\text{by Definition 4.6} \\
\alpha(D_Q \uparrow \omega) \setminus (R \cup S). &&&
\end{aligned}
$$

∎

To prove Lemma 4.10 we introduce some preliminary terminology and definitions. Let a *computation tree* be defined as a finite tree whose nodes are labeled by either $\square$ or by pairs (A, s_A), where A is a ground atom and s_A is a set of predicate symbols. To ease the presentation, we will henceforth identify the nodes of a computation tree with their labels. A leaf of a computation tree T is said to be *open* if (its label) is not $\square$, *closed* otherwise; a node is *closed* if $\square$ is its unique child.

Next we introduce the definition of *unfolding tree*: intuitively, an unfolding tree for a program P is a computation tree T such that if (A, s) is the root and $(B_1, \emptyset), \ldots, (B_k, \emptyset)$ are the open leaves of T, then T represents the same computation as the one encoded, in the fixed point $D_P \uparrow \omega$, by the cs-clause $A \leftarrow s \square B_1, \ldots, B_k$. Here we extend this idea by associating unfolding tree to sets of cs-clauses (hence, to cs-interpretations) rather than simply to differential programs. To do that, we will work with the extension of D_P to sets of cs-clauses (see remark after Definition 4.8) and write P to denote, in general a set of cs-clauses rather than simple clauses. Also, since in every cs-interpretation the body of a cs-clause is viewed as a set of atoms, we will henceforth assume that also the body of the clauses in the ground extension of a program are set of atoms. A formal definition of unfolding trees is as follows.

Let the depth of a computation tree be defined as the maximum length of a root-leaf path in the tree. Let also $root(T)$ denote the root of any computation tree T and let $susp(T)$ denote the set $\{B \mid (B, \emptyset) \text{ is an open leaf of } T\}$.

We first define the set $T^0_{\mathtt{Id}}(P)$ of *identical trees* of depth 0 for P as follows:

$$T^0_{\mathtt{Id}}(P) = \{T \mid \quad T \text{ is a computation tree consisting of a single node} \atop \qquad \text{labeled by } (A, \emptyset) \text{ where } Pred(A) \subseteq Open(P) \qquad \}.$$

Moreover, we define the set of *identical trees* of any depth as follows

$$T_{\mathtt{Id}}(P) = \{T \mid \quad T \text{ is a computation tree s.t. } root(T) = (A, \emptyset) \atop \qquad Pred(A) \subseteq Open(P) \text{ and } susp(T) = \{A\} \qquad \}.$$

Note that $T_{\mathtt{Id}}(P) \supseteq T^0_{\mathtt{Id}}(P)$.

Now we define the set $T_n(P)$ of the unfolding trees for P of depth $\leq n$. $T_n(P)$ is defined by induction as the smallest set satisfying the following conditions:

$(n = 0)$ $T_0(P) = \emptyset$.

$(n > 0)$ For every cs-clause $A \leftarrow s \,\square\, \{B_1, \dots, B_k\} \in Ground(P)$:

      (i) if $k = 0$ then there exists $T \in T_n(P)$ such that $root(T) = (A, s)$ and (A, s) is a close node;

      (ii) if $k > 0$ and for all $j \in [1 \dots k]$, there exists a tree T_j in $T_{n-1}(P) \cup T^0_{\mathtt{Id}}(P)$ with $root(T_j) = (B_j, s_j)$, then there exists $T \in T_n(P)$ such that

        - $root(T) = (A, s \cup s_0 \cup s_1 \dots \cup s_k)$, and
           $s_0 = \{Pred(B_j) \mid j \in [1 \dots k], Pred(B_j) \subseteq \text{DYN and } T_j \notin T_{\mathtt{Id}}(P)\}$
        - $T_1, \dots, T_k$ are the principal subtrees of T.

In case (ii) we say that T is expanded with the clause $A \leftarrow s \,\square\, \{B_1, \dots, B_k\}$. Finally, the set $T(P)$ of all the unfolding trees for a differential program P is defined as follows:

$$T(P) = \bigcup_{n \geq 0} T_n(P).$$

The correspondence between the unfolding trees in $T(P)$ and the cs-clauses of $D_P \!\uparrow\! \omega$ is established as follows. Let $clause(T)$ in $\mathcal{C}$ be the *cs-clause associated with T* defined as follows: if $root(T) = (A, s)$, then

$$clause(T) = A \leftarrow s \,\square\, susp(T).$$

Clearly an unfolding tree provides a by far more concrete encoding of a computation than the corresponding cs-clauses in $D_P \!\uparrow\! \omega$: in fact, it is easy to verify that the same cs-clause may be associated to several different unfolding trees. We have, however the following result.

Lemma A.2 Denote with $clauses(T)$ the set $\{clause(T) \mid T \in T\}$ for every set of unfolding trees T. Then:

$$D_P \uparrow \omega = clauses(T(P)).$$

PROOF. The claim follows easily by induction on n: we can prove that for every $n \geq 0$, $D_P \!\uparrow\! n = clauses(T_n(P))$. ∎

In the proof of Lemma 4.10 we will need a number of operations on unfolding trees that are introduced below. We start with an operation that builds an unfolding tree by expanding a given unfolding tree at some of its open leaves.

Definition A.3 Let P be a differential program, $T \in T(P)$ an unfolding tree such that $root(T) = (A, s_A)$ and $susp(T) = \{B_1, \ldots, B_k\}$. Let also I, J be a partition of $[1 \ldots k]$ such that for every $i \in I$, $T_i \in T(P)$ and $root(T_i) = (B_i, s_i)$. Then we define $T \oplus \{T_i\}_{i \in I}$ to be the unfolding tree such that:

$$clause(T \oplus \{T_i\}_{i \in I}) = A \leftarrow (s_A \cup s_{susp}) \,\square\, \bigcup_{i \in I} susp(T_i) \cup \{B_j \mid j \in J\}$$

where $s_{susp} = \bigcup_{i \in I} s_i \cup \{Pred(B_i) \mid i \in I, Pred(B_i) \subseteq \text{DYN} \text{ and } T_i \notin T_{\texttt{Id}}(P)\}$.

That $T \oplus \{T_i\}_{i \in I}$ is a well-defined unfolding tree in $T(P)$ follows from the definition of unfolding tree. Next we define an operation that builds an unfolding tree from a given unfolding tree by deleting some of its subtrees.

Definition A.4 Let P be a differential program, $T \in T(P)$ and $N = \{n_1, \ldots, n_h\}$ be a set of internal nodes of T such that $n_i = (B_i, s_i)$, $Pred(B_i) \subseteq Open(P)$ and $\forall i, j \in [1 \ldots h]$, $i \neq j \Rightarrow n_i$ is not an ancestor of n_j. Let also $\{T_i\}_{i \in [1 \ldots h]}$ be the set of the subtrees of T such that $root(T_i) = n_i$. Then we define $T \setminus \{T_i\}_{i \in [1 \ldots h]}$ to be the computation tree such that:

$$clause((T \setminus \{T_i\}_{i \in [1 \ldots h]}) \oplus \{T_i\}_{i \in [1 \ldots h]}) = clause(T).$$

Again, that $T \setminus \{T_i\}_{i \in [1 \ldots h]}$ is a well-defined unfolding tree follows from the definition of unfolding tree and from Definition A.3.

Among all the unfolding trees which are associated to the same cs-clause we identify and isolate those which satisfy the following condition.

Definition A.5 **(Non Redundant Trees)** Call *tautological* every unfolding tree T such that *clause(T)* is a tautology. We say that T is *non redundant* if and only if:

- T is non-tautological and
- for all proper subtrees T' of T, either $T' \in T^0_{\texttt{Id}}(P)$ or T' is non-tautological.

The following Lemma shows that, given a differential program P, if we consider a clause in $\alpha(clauses(T(P)))$ it is not restrictive to limit our attention to non redundant trees. Given a cs-clause $\texttt{c} = A \leftarrow s \,\square\, \mathbf{B}$, denote with $\Delta(\texttt{c})$ the cs-clause $A \leftarrow (s \setminus Pred(A)) \,\square\, \mathbf{B}$.

Lemma A.6 Let P be a differential program and $\texttt{c} \in \alpha(clauses(T(P)))$. Then there exists a non redundant tree $T \in T(P)$ such that $\texttt{c} = \Delta(clause(T))$.

PROOF. From the definition of the α abstraction, it follows that for every cs-clauses $\texttt{c}$, if $\alpha(\{\texttt{c}\}) \neq \emptyset$, then $\alpha(\{\texttt{c}\}) = \{\Delta(\texttt{c})\}$. Then, since $\texttt{c} \in \alpha(clauses(T(P)))$, there must exist at least one $T^\star \in T(P)$ such that $\texttt{c} = \Delta(clause(T^\star))$.

Now assume that $N = \{n_1, \ldots, n_h\}$ is the maximal set of nodes in $T^\star$ such that for every $n_i \in N$ T_i the subtree of $T^\star$ rooted at n_i, is tautological and none of the trees rooted at the ancestors of n_i are tautological. Let $T = T^\star \setminus \{T_i\}_{i \in [1 \ldots h]}$. Clearly, T is non redundant and $\Delta(clause(T))$ subsumes $\Delta(clause(T^\star))$. But subsumption cannot be strict since $\texttt{c} = \Delta(clause(T^\star)) \in \alpha(clauses(T(P)))$. Hence $\texttt{c} = \Delta(clause(T))$. ∎

Lemma A.7 Let P be a differential program, T be a non redundant unfolding tree in $T(P)$ and let $T_B \notin T_{\mathtt{Id}}^0(P)$ be a proper subtree of T, with $root(T_B) = (B, s_B)$ and $Pred(B) \not\subseteq$ INT. If $\alpha(\{clause(T)\}) \neq \emptyset$ then also $\alpha(\{clause(T_B)\}) \neq \emptyset$.

PROOF. Let $root(T) = (A, s)$. By definition of unfolding tree, (i) $s \supseteq s_B \cup (Pred(B) \cap \text{DYN})$ and (ii) $susp(T) \supseteq susp(T_B)$. Now consider the operators involved in the definition of the abstraction α.

Take ι first. By hypothesis $Pred(B) \not\subseteq$ INT and therefore $\iota(\{clause(T_B)\}) = \{clause(T_B)\}$.

Now consider δ: by hypothesis, (iii) $\alpha(\{clause(T)\}) \neq \emptyset$ and thus (iv) $s \cap Pred(susp(T_B)) = \emptyset$. Then we have:

- $(s_B \setminus (Pred(B))) \cap Pred(susp(T_B)) = \emptyset)$, immediately from from (i), (ii) and (iv).

- $(Pred(B) \not\subseteq (Pred(susp(T_B)) \cap \text{DYN})$, by contradiction.
  Assume that $Pred(B) \subseteq Pred(susp(T_B)) \cap \text{DYN}$. Then, from by (i) and (ii), we have $s \cap Pred(susp(T)) \neq \emptyset$ which contradicts (iv).

By definition A.5, since T is non redundant, then T_B is a non tautological unfolding tree. Hence, if $\mathsf{c} \in \delta(\iota(\{clause(T_B)\}))$, then c is not a tautology and therefore $wcf(\delta(\iota(\{clause(T_B)\}))) = \alpha(\{clause(T_B)\}) \neq \emptyset$. ∎

Now consider two programs P and Q that satisfy the hypothesis of Lemma 4.10. The next operation builds an unfolding tree for P (or Q) out of a given unfolding tree for the composite program $P \cup Q$. We need first a definition.

Definition A.8 Let P and Q be two differential programs and let T be an unfolding tree in $T_n(P \cup Q)$. We say that a node n in T is an *interface node* if it is an internal node of T and the subtree rooted at the father of n has been expanded with a clause in P (resp. Q) iff the subtree rooted at n has been expanded with a clause in Q (resp. P).

Definition A.9 Let P and Q be two differential programs such that P *isa* Q is a well-formed hierarchy and $\nabla_P \cap Defn(Q) = \emptyset$. Let also T be an unfolding tree in $T_n(P \cup Q)$. Assume $N = \{n_1, \ldots, n_h\}$ is the maximal set of interface nodes in T such that for every $n_i \in N$ none of the ancestors of n_i in T is an interface node. If $\{T_i\}_{i \in [1\ldots h]}$ is the set of the subtrees of T rooted at the nodes of N then we define:

$$T_{up} = T \setminus \{T_i\}_{i \in [1\ldots h]} \quad \text{and} \quad Rest(T) = \{T_i\}_{i \in [1\ldots h]}.$$

First note that T_{up}, and consequently, $Rest(T)$ are unique for every unfolding tree T, being the set N maximal. That T_{up} is a well-defined unfolding tree follows from the following lemma.

Lemma A.10 Let P and Q be two differential programs such that P *isa* Q is a well-formed hierarchy and $\nabla_P \cap Defn(Q) = \emptyset$. Moreover, let T be an unfolding tree in $T(P \cup Q)$, $n = (A, s)$ an interface node in T, T_n the subtree of T rooted at n, c the clause used to expand T_n. Then the following property holds: if $\mathsf{c} \in Q$ then $Pred(A) \subseteq Open(P)$, otherwise $Pred(A) \subseteq Open(Q)$.

PROOF. Since n is an interface node and P *isa* Q is a well-formed hierarchy, $Pred(A) \not\subseteq$ INT. If it is extensible or dynamic we immediately have the result. As for static predicates, when $\mathsf{c} \in Q$, we have $Pred(A) \subseteq \text{STAT}(P) \cap Defn(Q)$: since $\nabla_P \cap Defn(Q) = \emptyset$ by hypothesis, we conclude that $Pred(A) \not\subseteq Defn(P)$ and thus $Pred(A) \subseteq Open(P)$. When $\mathsf{c} \in P$ we prove, by contradiction, that $Pred(A) \not\subseteq \text{STAT}(Q)$. Assume $Pred(A) \subseteq \text{STAT}(Q)$, then $Pred(A) \subseteq Defn(Q)$, by well-formedness; from $\mathsf{c} \in P$, clearly $Pred(A) \subseteq \text{STAT}(P) \cap Defn(P)$ which contradicts the hypothesis $\nabla_P \cap Defn(Q) = \emptyset$. ∎

Note, in particular, that if T_{up} is expanded at the root with a clause from P then $T_{up} \in T(P)$; otherwise it $T_{up} \in T(Q)$. In fact we can prove something more as the following lemma shows.

Lemma A.11 Let P and Q be two differential programs such that P *isa* Q is a well-formed hierarchy and $\nabla_P \cap Defn(Q) = \emptyset$. Let also T be an unfolding tree in $T(P \cup Q)$ and T_{up} be the unfolding tree of Definition A.9.
If T_{up} is not tautological and $\alpha(\{clause(T)\}) \neq \emptyset$ then there exists a clause $\mathsf{c} \in F_l(P) \cup F(Q)$ such that c subsumes $\Delta(clause(T_{up}))$.

PROOF. Denote with R the program (either P or Q) such that $T_{up} \in T(R)$. Let also $\Delta(clause(T)) = A \leftarrow s \;\square\; susp(T)$ and $\Delta(clause(T_{up})) = A \leftarrow s_{up} \;\square\; susp(T_{up})$. Being $T_{up} \in T(R)$, by Lemma A.2, $clause(T_{up}) \in D_R \uparrow \omega$. We first prove that $\alpha(\{clause(T_{up})\}) = \{\Delta(clause(T_{up}))\}$.

Consider ι first. Since $\alpha(\{clause(T)\}) \neq \emptyset$, $Pred(A) \not\subseteq \text{INT}$ and hence $\iota(\{clause(T_{up})\}) = \{clause(T_{up})\}$.

Now consider δ: we show that $\delta(\{clause(T_{up})\}) = \{\Delta(clause(T_{up}))\}$. Since T_{up} is non-tautological by hypothesis, clearly $Pred(A) \not\subseteq Pred(susp(T_{up})) \cap \text{DYN}$. Hence, we need only to show that $s_{up} \cap Pred(susp(T_{up})) = \emptyset$.
By contradiction, assume there exists B such that $Pred(B) \subseteq Pred(Susp(T_{up})) \cap s_{up}$: clearly, $Pred(B) \subseteq Defn(R)$. Now we have two possible cases: if the node corresponding to B is an interface node in T, then $Pred(B) \subseteq \nabla(P) \cap Defn(Q)$ which is a contradiction. Otherwise, if $B \in susp(T)$, then $Pred(B) \subseteq Pred(Susp(T)) \cap s_{up}$ and hence $Pred(B) \subseteq Pred(Susp(T)) \cap s$ being $s_{up} \subseteq s$. But this is a contradiction because it implies that $\alpha(\{clause(T)\}) = \emptyset$ against our inital hypothesis.

Finally, since T_{up} is not tautological by hypothesis, then

$$\alpha(\{clause(T_{up})\}) = wcf(\delta(\iota(\{clause(T_{up})\}))) = \{\Delta(clause(T_{up}))\}$$

Since $T_{up} \in T(R)$, from Lemma A.2, $clause(T_{up}) \in D_R \uparrow \omega$. Furthermore, since $\alpha(\{clause(T_{up})\}) = \{\Delta(clause(T_{up}))\}$, there exists a clause $\mathsf{c} \in F(R)$ such that c subsumes $\Delta(clause(T_{up}))$. Now, if $R = Q$, then clearly $\mathsf{c} \in F_l(P) \cup F(Q)$ and we are done. Otherwise, when $R = P$, we need to show that $\mathsf{c} \in F_l(P)$. To see this we prove that $Pred(susp(T_{up})) \cap \text{STAT}(P) \subseteq Defn(F(Q))$.

Take $B \in susp(T_{up})$ such that $Pred(B) \subseteq \text{STAT}(P)$. Being $Pred(susp(T)) \subseteq Open(P \cup Q)$ and P *isa* Q well formed, $B \notin susp(T)$. Then, (B, s_B) is the root of one of the unfolding trees in $Rest(T)$. Let T^B be that tree: then $T_{up}^B \in T(Q)$ and hence, by Lemma A.2 $clause(T_{up}^B) \in D_Q \uparrow \omega$. Now observe that $Pred(B) \not\subseteq Pred(susp(T_{up}^B))$ since $Pred(B) \not\subseteq Open(Q)$, being P *isa* Q well-formed and $Pred(B) \subseteq \text{STAT}(Q)$. Then, there exists a clause in $\alpha(D_Q \uparrow \omega) = F(Q)$ which subsumes $\Delta(clause(T_{up}^B))$ and hence defines $Pred(B)$. ∎

We can finally prove Lemma 4.10.

Lemma 4.10 Let P and Q be differential programs such that P *isa* Q is a well-formed hierarchy and $\nabla_P \cap Defn(Q) = \emptyset$. Moreover let $F_l(P)$ be defined according to Definition 4.8. Then

$$F(P \cup Q) = F(F_l(P) \cup F(Q)).$$

PROOF. By Lemma A.2 we have to prove that:

$$\alpha(clauses(T(P \cup Q))) = \alpha(clauses(T(F_l(P) \cup F(Q)))). \tag{1}$$

The proof is in three steps:

Step 1: We first show that for every non redundant tree $T \in T(F_l(P) \cup F(Q))$, if $\alpha(\{clause(T)\}) \neq \emptyset$ then there exists an unfolding tree $T^\star$ in $T(P \cup Q)$ such that $\Delta(clause(T^\star))$ subsumes $\Delta(clause(T))$.

Step 2: Then we prove that for every non redundant tree $T \in T(P \cup Q)$, if $\alpha(\{clause(T)) \neq \emptyset$ then there exists an unfolding tree $T^\star$ in $T(F_l(P) \cup F(Q))$ such that $\Delta(clause(T^\star))$ subsumes $\Delta(clause(T))$.

Step 3: We conclude that $\alpha(clauses(T(F_l(P) \cup F(Q)))) = \alpha(clauses(T(P \cup Q)))$ proceeding by contradiction.

Proof of Step 1. The proof is by the induction on the depth of T. The base case, for $n = 0$, is trivial because there is no tree T satisfying the hypothesis. Let then $n \geq 1$ be the depth of T. Assume that $\mathsf{c} = A \leftarrow s_{\mathsf{c}} \,\square\, \{B_1, \ldots, B_k\} \in F_l(P) \cup F(Q)$ be the clause used to expand T at its root, and let I, J be a partition of $[1 \ldots k]$ such that the set $\{T_i \mid i \in I\} \subseteq T_{n-1}(F_l(P) \cup F(Q)) \setminus T^0_{\mathtt{Id}}(F_l(P) \cup F(Q))$ while the set $\{T_j \mid j \in J\} \subseteq T^0_{\mathtt{Id}}(F_l(P) \cup F(Q))$.

Clearly, c is either in $F_l(P)$ or in $F(Q)$. If $\mathsf{c} \in F_l(P)$, then $\mathsf{c} \in \alpha(D_Q \uparrow \omega)$ and, by Lemma A.2 $\mathsf{c} \in \alpha(clauses(T(P)))$. Hence, there exists $T_{\mathsf{c}} \in T(P)$ such that $\mathsf{c} = \Delta(clause(T_{\mathsf{c}}))$. For the the same reasoning, if $\mathsf{c} \in F(Q)$, then $T_{\mathsf{c}} \in T(Q)$. Now consider the principal subtrees of T.

Since $Open(F_l(P) \cup F(Q)) \subseteq Open(P \cup Q)$, then $T_j \in T^0_{\mathtt{Id}}(P \cup Q)$, for all $j \in J$. Hence, $T_{\mathsf{c}} \in T(P \cup Q)$.

As for the set $\{T_i \mid i \in I\} \subseteq T_{n-1}(F_l(P) \cup F(Q)) \setminus T^0_{\mathtt{Id}}(F_l(P) \cup F(Q))$. By Definition A.5, every tree in this set is non redundant. Furthermore, since, clearly, $\mathrm{INT}(F_l(P) \cup F(Q)) = \emptyset$, for all $i \in I$ if $clause(T_i) = A_i \leftarrow s_i \,\square\, susp(T_i)$, then $Pred(A_i) \not\subseteq \mathrm{INT}$. Then, letting d_i denote $clause(T_i)$, by Lemma A.7 $\alpha(\{\mathsf{d}_i\}) \neq \emptyset$.

Now we can apply the inductive hypothesis on the T_is: as a consequence, for every $i \in I$, there exists $T_i^\star$ in $T(P \cup Q)$ such that $clause(T_i^\star) = \mathsf{d}_i^\star = A_i \leftarrow s_i^\star \,\square\, susp(T_i^\star)$ and $\mathsf{d}_i^\star$ subsumes d_i.

By Definition A.3, $T^\star = T_{\mathsf{c}} \oplus \{T_i^\star\}_{i \in I}$ is an unfolding tree in $T(P \cup Q)$ such that:

$$clause(T^\star) = A \leftarrow s^\star_{susp} \,\square\, \bigcup_{i \in I} susp(T_i^\star) \cup \{B_i \mid i \in J\}$$

where

$$s^\star_{susp} = s' \cup \{s_i^\star \mid i \in I\} \cup (\{Pred(B_i) \mid i \in I\} \cap \mathrm{DYN})$$

and

$$s' \setminus \{Pred(A)\} = s_{\mathsf{c}}.$$

Finally, let $s = s_{\mathsf{c}} \cup \{s_i \mid i \in I\} \cup (\{Pred(B_i) \mid i \in I\} \cap \mathrm{DYN})$. Since $\mathsf{d}_i^\star$ subsumes d_i, by definition, $s_i^\star \subseteq s_i$ and $susp(T_i^\star) \subseteq susp(T_i)$. Now we can conclude as follows:

$$
\begin{aligned}
\Delta(clause(T^\star)) \quad &= \quad A \leftarrow (s^\star_{susp} \setminus Pred(A)) \,\square\, \bigcup_{i \in I} susp(T_i^\star) \cup \{B_j \mid j \in J\} \\
&\text{subsumes} \quad A \leftarrow (s \setminus Pred(A)) \,\square\, \bigcup_{i \in I} susp(T_i) \cup \{B_j \mid j \in J\} \\
&= \quad \Delta(clause(T)).
\end{aligned}
$$

Therefore, $T^\star \in T(P \cup Q)$ and $\Delta(clause(T^\star))$ subsumes $\Delta(clause(T))$.

Proof of Step 2. The proof is by induction on the depth of T. The base case is trivial. Then, assume that T has depth $n \geq 1$ and let T_{up} be the unfolding tree associated to T. Let also $Rest(T) = \{T_i \mid i \in I\}$ and $\{(B_j, \emptyset) \mid j \in J\}$ be the set of open leaves which are common to T and T_{up}. By Lemma A.11, if T_{up} is not tautological, there exists a clause $\mathsf{c} = A \leftarrow s \square \mathbf{B} \in F_l(P) \cup F(Q)$ such that c subsumes $\Delta(clause(T_{up}))$. Let, for every $i \in I$, $\mathsf{d}_i = clause(T_i) = A_i \leftarrow s_i \square susp(T_i)$.

Since T is non redundant, then so is T_i for every $i \in I$. Furthermore, by definition of interface nodes and since P isa Q is a well-formed hierarchy, for every $i \in I$, $Pred(A_i) \not\subseteq \textsc{int}$ and then, by Lemma A.7, $\alpha(\{clause(T_i)\}) \neq \emptyset$. Therefore, by the inductive hypothesis, there exists $T_i^\star$ in $T(P \cup Q)$ such that $\Delta(clause(T_i^\star))$ subsumes $\Delta(clause(T_i))$. Now, we have two possibilities:

(i) if T_{up} is tautological, since T is non redundant (and therefore $clause(T)$ is not a tautology), there exists $l \in I$ such that $B_l = A$. Then let $T^\star = T_l^\star$.
(ii) Otherwise let $T^\star$ be the unfolding tree in $T(F_l(P) \cup F(Q))$ obtained by expanding $\mathsf{c} = A \leftarrow s \square \mathbf{B}$, with $T_i^\star$, for all $i \in I$ such that $B_i \in \mathbf{B}$, and tautological trees in $T_{\mathtt{Id}}^0(F_l(P) \cup F(Q))$, with roots labeled by $(B_j, \emptyset)$, for all $j \in J$ such that $B_j \in (\mathbf{B} \setminus \{B_i \mid i \in I\})$.

By repeating on $T^\star$ the same reasoning used in Step 1, we obtain:

$$\Delta(clause(T^\star)) \text{ subsumes } \Delta(clause(T)).$$

Proof of Step 3. Finally we prove that

$$\mathsf{c} \in \alpha(clauses(T(F_l(P) \cup F(Q)))) \iff \mathsf{c} \in \alpha(clauses(T(P \cup Q)))$$

The reasoning is symmetric in the two directions. Here we proof only the case $\Rightarrow$. The proof is by contradiction. Assume that $\mathsf{c} \notin \alpha(clauses(T(P \cup Q)))$.

Observe that, by Lemma A.6, if $\mathsf{c} \in \alpha(clauses(T(F_l(P) \cup F(Q))))$ then there exists a tree $T \in T(F_l(P) \cup F(Q))$ such that $\mathsf{c} = \Delta(clause(T))$ and T is non redundant. Clearly, for any such T, $\alpha(\{clause(T)\}) \neq \emptyset$.

Let $T^\star$ be the unfolding tree in $T(P \cup Q)$ built in Step 1: then $\Delta(clause(T^\star))$ subsumes c. Now, since $\delta(\iota(\{clause(T^\star)\})) = \{\Delta(clause(T^\star))\}$, there exists $\tilde{T} \in T(P \cup Q)$ such that

(i) $\Delta(clause(\tilde{T})) \in \alpha(clauses(T(P \cup Q)))$ and   (ii) $\Delta(clause(\tilde{T}))$ subsumes $\Delta(clause(T^\star))$

Since $\Delta(clause(T^\star))$ subsumes c and $\mathsf{c} \notin \alpha(clauses(T(P \cup Q)))$, by (ii),

$$\Delta(clause(\tilde{T})) \text{ strictly subsumes } \mathsf{c}. \tag{2}$$

By (i), $\{\Delta(clause(\tilde{T}))\} \neq \emptyset$, then, by Step 2, there exists $\tilde{T}^\star$ in $T(F_l(P) \cup F(Q))$ such that

$$\Delta(clause(\tilde{T}^\star)) \text{ subsumes } \Delta(clause(\tilde{T})). \tag{3}$$

Then, by (3) and (2) we obtain

$$\Delta(clause(\tilde{T}^\star)) \text{ strictly subsumes } \mathsf{c}$$

which is a contradiction since $\tilde{T}^\star$ in $T(F_l(P) \cup F(Q))$ and $\mathsf{c} \in \alpha(clauses(T(F_l(P) \cup F(Q))))$. $\blacksquare$