Abstract. This paper presents a logic framework for the incremental inductive synthesis of Datalog theories. It allows us to cast the problem as a process of abstract diagnosis and debugging of an incorrect theory. This process involves a search in a space, whose algebraic structure (conferred by the notion of object identity) makes easy the definition of algorithms that meet several properties that are deemed as desirable from the point of view of the theoretical computer science. Such algorithms embody two refinement operators, one for generalizing Datalog clauses which do not cover positive examples, and the other one for specializing Datalog clauses that explain negative examples. The former is based on the concept of least general generalization under object identity, while the latter takes advantage of the algebraic structure of the search space to meet the fundamental properties of local finiteness, properness and completeness (ideality). These algorithms have been implemented in a new version of INCR/H, a system for the incremental inductive synthesis of Datalog theories, whose main characteristic consists of the capability of autonomously performing a representation change, that allows it to extend the search to the space of Datalog ¬ clauses, when no correct theories exist in the space of Datalog clauses. Experimental results in the area of electronic document understanding show that INCR/H is able to cope effectively and efficiently with this real-world task.

1. Introduction

A logical theory can be viewed as a set of conditions, expressed in a logical language, that are necessary and sufficient to explain a number of observations in a given environment. In addition to the capability of explaining past events, the usefulness of a theory relies on its ability of predicting future situations in the same environment. If we assume that the only source of knowledge available is represented by a set of previously classified observations and no prior knowledge can be exploited, the process of formulating a new theory is bound to be progressive. Starting from contingent observations, it is not possible to infer concept definitions that are universally regarded as correct. The validity of the theory itself extends to the available knowledge. Conversely, new observations can point out the inadequacies in the current formulation of the concepts. In such a case, the theory is incorrect and a suitable process of theory revision should be activated.

A theory may be incorrect because, given a new observation, one of the following cases occurs:

- this observation is erroneously explained by the theory, thus the theory is too general and needs to be specialized;
- this observation is erroneously not explained by the theory, thus the theory is too specific and needs to be generalized.

In this paper, we address these problems in a logic framework. The solutions that we propose require to perform a process of abstract diagnosis and debugging of the theory in order to restore its correctness. Specifically, the debugging of the theory is cast as a search for either a specialization (downward refinement) or a generalization (upward refinement) of that part of the theory detected as source of incorrectness by the diagnosis step. This search aims at finding a minimal refinement of a theory [Wrobel, 93]. The same goal is pursued in the field of belief revision [Gärdenfors & Rott, 92] and theory contraction [Fuhrmann, 91]. Indeed, in a logic-constrained belief revision approach, a contraction of a belief set with respect to (wrt) a new fact consists of a peculiar belief change, which requires the retraction of the information causing the violation of consistency, when this property is regarded as an integrity constraint.

Formulating logical theories from facts is the ultimate objective of concept learning. In this area, a theory consists of a set of hypotheses, a hypothesis is a concept definition and observations are called examples. An example that should be explained by a theory is called positive, an example that should be refuted is called negative.

Initially, the research efforts in this area centred on the analysis and development of inductive systems that synthesize a logical theory in a batch way. These systems start from an empty theory and stop the inductive process when the current set of hypotheses is able to explain all the available examples. When new evidence contradicts the synthesized theory, the whole process must be repeated, taking no advantage of the previous version of the hypotheses. Such a drawback can
be overcome by means of incremental inductive systems. These systems are able to refine and restate a theory in an 
incremental way, thus the previously generated hypotheses are not completely rejected, but they are taken as the starting 
point of a search process whose goal consists of a new theory that explains both old and new observations. Incremental 
synthesis of theories is necessary in several cases, like changing world, sloppy modeling and selection bias [Wrobel, 94], 
even though problems might arise with recursive theories.

This paper addresses the problem of incremental synthesis of theories in a logic framework, that allows us to cast this 
problem as a search in a space whose algebraic structure is now thoroughly understood [Semeraro et al., 94], and that 
makes easy the definition of search algorithms which meet many properties that are deemed as desirable from the point 
of view of the mathematical theory of computation (e.g., non-termination) [Semeraro et al., 94], as well as from that of 
the theory of the computational complexity [Esposito et al., 96; Semeraro et al., 96].

The plan of the paper is as follows. In the next section, the logical language adopted to represent both the synthesized 
theory and the examples is briefly introduced. Section 3 recalls the logic framework, based on the notion of object identity, 
within which the problem of debugging an incorrect theory is addressed. Section 4 introduces the basic notions regarding 
incremental inductive synthesis of logical theories, while Sections 5 and 6 present the refinement operators that have been 
developed and implemented in a new version of INCR/H [Semeraro et al., 95], an incremental system that can inductively 
synthesize and correct logical theories. The results of the application of INCR/H to the real-world problem of document 
understanding are shown in Section 7.

2. The Representation Language

Henceforth, we refer to [Lloyd, 87] for what concerns the basic definitions of a substitution, positive and negative literal, 
clause, definite and program clause, and normal program. We will indifferently use the set notation and the Prolog 
notation for clauses. Given a first-order clause $C$, $\text{vars}(C), \text{consts}(C)$ and $|C|$ denote respectively the set of the variables, 
the set of the constants and the number of literals occurring in $C$. By logical theory we mean a set of hypotheses; by hypothesis 
we mean a set of program clauses with the same head. In the paper, we are concerned exclusively with logical theories 
expressed as hierarchical programs, that is, as (non-recursive) programs for which it is possible to find a level mapping 
[Lloyd, 87] such that, in every program clause $P(t_1, t_2, \ldots, t_n) \leftarrow \overline{L}_1, \overline{L}_2, \ldots, \overline{L}_n$, the level of every predicate symbol occurring 
in the body is less than the level of $P$. Another constraint on the representation language is that, whenever we write about 
clauses, we mean Datalog linked clauses. Here, we refer to [Ceri et al., 90; Kanellakis, 90] for the basic notions about 
Datalog and its extensions Datalog$^+$ and Datalog$^-$. A definition of linked clause is the following [Helft, 87]. A Horn 
clause is linked if all of its literals are; a positive literal is linked if at least one of its arguments is; an argument of a literal 
is linked if either the literal is the head of the clause or another argument in the same literal is linked. An instance of a 
linked clause is $C = P(x) \leftarrow Q(x, y), Q(y, z)$. Conversely, the clauses $\overline{D} = C \cup \{\neg R(v, w)\}$ and $\overline{F} = C \cup \{\neg R(v, w)\}$ are 
not linked. Indeed, the literal $Q(y, z)$ is not linked in $D$, whereas $\neg R(v, w)$ is not linked in $F$.

The differences existing between examples and hypotheses are the following.
• Each example is represented by one ground clause with a unique literal in the head.
• Each hypothesis is a set of program clauses with the same head.

An example $E$ is positive for a hypothesis $H$ if its head has the same predicate letter and sign as the head of the clauses 
in $H$. The example $E$ is negative for $H$ if its head has the same predicate, but opposite sign. Thus, more precisely, a negative 
example is a general Horn clause [Grant & Subrahmanian, 95]. Another observation concerning the representation 
language is that no predicate invention is performed.

3. Object Identity and $\theta$-subsumption

Here, we introduce a logic language, called Datalog$^{OI}$, which is an instance of constraint logic programming [Jaffar & 
Maher, 94]. The basic notion of Datalog$^{OI}$ is that of object identity. We recall the definition of object identity, previously 
given in [Semeraro et al., 94] for a first order logic that is both function-free and constant-free, and later generalized to 
a full first order logic [Esposito et al., 96; Semeraro et al., 96].

Definition 1 (Object Identity) Within a clause, terms denoted with different symbols must be distinct.

This notion is the basis for the definition of both an equational theory for Datalog clauses and a quasi-ordering upon 
them. In Datalog, the adoption of the object identity assumption can be viewed as a method for building an equational 
theory into the ordering as well as into the inference rules of the calculus (resolution, factorization and paramodulation) 
[Plotkin, 72]. Such equational theory is very simple, since it consists of just one further axiom schema (axiom 13 in boldface), 
in addition to the set of the axioms of Clark's Equality Theory (CET) [Lloyd, 87]:

1. $\forall (x = x)$
2. $\forall (x = y \Rightarrow y = x)$
3. $\forall (x = y \land y = z \Rightarrow x = z)$
∀ (x₁ = y₁) ∧ ... ∧ (xₙ = yₙ) ⇒ f(x₁, ..., xₙ) = f(y₁, ..., yₙ), for each function symbol f (substitutivity)

(5) ∀ (x₁ = y₁) ∧ ... ∧ (xₙ = yₙ) ⇒ P(x₁, ..., xₙ) ⇔ P(y₁, ..., yₙ), for each predicate symbol P (including =) (substitutivity)

(6) ∀ (f(x₁, ..., xₙ) = f(y₁, ..., yₙ)) ⇒ (x₁ = y₁) ∧ ... ∧ (xₙ = yₙ), for each function symbol f

(7) c ≠ d for all pairs c, d of distinct constants

(8) ∀ (f(x₁, ..., xₙ) ≠ g(y₁, ..., yₙ)), for all pairs f, g of distinct functional symbols

(9) ∀ (f(x₁, ..., xₙ) ≠ c), for each constant c and function symbol f

(10) ∀ (t[x] ≠ x), for each term t[x] containing the variable x and different from x

(11) ∀ (x₁ ≠ y₁) ∨ ... ∨ (xₙ ≠ yₙ) ⇒ f(x₁, ..., xₙ) ≠ f(y₁, ..., yₙ), for each function symbol f

(12) ∀ (f(x₁, ..., xₙ) ≠ f(y₁, ..., yₙ)) ⇒ (x₁ ≠ y₁) ∨ ... ∨ (xₙ ≠ yₙ), for each function symbol f

(13) t ≠ s ∈ body(C) for each clause C in L and for all pairs t, s of distinct terms that occur in C (OI)

where L denotes the language that consists of all the possible Datalog clauses built from a finite number of predicates.

Note that axioms (4), (6), (8), (9), (10), (11) and (12) are useless for Datalog (since they involve function symbols, that cannot occur in a Datalog clause). The (OI) axiom can be viewed as an extension of both Reiter's unique-names assumption [Reiter, 80] and axioms (7), (8) and (9) of CET to the variables of the language.

Under object identity assumption, the Datalog clause C = P(x) :- Q(x, x), Q(y, a) || [x ≠ y], [x ≠ a], [y ≠ a], where P, Q denote predicate literals, x, y variables, a is a constant and the inequations attached to the clause can be seen as constraints on its terms. These constraints are generated in a systematic way by the (OI) axiom. In addition, they can be dealt with in the same way as the other literals in the clause. Therefore, under object identity, any Datalog clause C generates a new Datalog clause C̄ consisting of two components, called core(C̄) and constraints(C̄), where core(C̄) = C and constraints(C̄) is the set of the inequations generated by the (OI) axiom, that is to say, constraints(C̄) = {t ≠ s | t, s ∈ terms(C), t, s distinct}. Therefore, Dataloḡ is a sublanguage of Datalog. Formally, a Dataloḡ program is made up of a set of Dataloḡ clauses of the form

Q(x₁, x₂, ..., xₙ) :- φ || I, n ≥ 0,

where φ and I are as in Datalog, and I is the set of inequations generated by the (OI) axiom. The symbol "||" means and just like ",", but is used for the sake of readability, in order to separate the predicates coming from the (OI) axiom from the rest of the clause.

Nevertheless, Dataloḡ has the same expressive power as Datalog. Indeed, it is possible to prove the following results (the proofs of all the results reported in the paper can be found in Appendix).

Proposition 1. ∀ C ∈ Datalog \exists C' = \{C₁, C₂, ..., Cₙ\} ⊆ Dataloḡ : T_c ↩ ω = T_c ↩ ω.

that is, for each Datalog clause we can find a set of Dataloḡ clauses which is equivalent to it.

Corollary 1. ∀ P ⊆ Datalog \exists P' ⊆ Dataloḡ : T_p ↩ ω = T_p ↩ ω.

that is, for any Datalog program we can find a Dataloḡ program equivalent to it.

Now, we can recall the ordering relation defined by the notion of θ-subsumption under object identity — θ̄̄-subsumption — upon the set of Datalog clauses [Esposito et al., 96; Semeraro et al., 96]. The following definition extends to Datalog the definition given in [Semeraro et al., 94] for constant-free (other than function-free) logic languages.

Definition 2 (θ̄̄-subsumption ordering) Let C, D be two Datalog clauses. We say that D θ̄̄-subsumes C under object identity (D θ̄̄C) if and only if (iff) there exists a substitution σ such that (s.t.) D̄̄ = C̄̄.

In such a case, we say that D is more general than or equivalent to C (D is an upward refinement of C and C is a downward refinement of D) under object identity and we write C ≤̄̄D. We write C <̄̄D when C ≤̄̄D and not(D ≤̄̄C) and we say that D is more general than C (D is a proper upward refinement of C) or C is more specific than D (C is a proper downward refinement of D) or D properly θ̄̄-subsumes C. We write C ~̄̄D, and we say that C and D are equivalent clauses under object identity, when C ≤̄̄D and D ≤̄̄C.

Like θ-subsumption, θ̄̄-subsumption induces a quasi-ordering upon the space of the Datalog clauses, as stated by the following result.

Proposition 2. Let C, D, E be Datalog clauses. Then:

(a) C ≤̄̄D

(b) C ≤̄̄D and D ≤̄̄E ⇒ C ≤̄̄E

A characterization of the notion of θ̄̄-subsumption is provided by the following proposition.

Proposition 3. Let C, D be two Datalog clauses,

C ≤̄̄D ⇔ ∃ σ s.t. core(D̄̄), σ ⊆ core(C̄̄) and constraints(D̄̄), σ ⊆ constraints(C̄̄)

3
4. Incremental Inductive Synthesis Basics

Generally, the canonical inductive paradigm requires the fulfillment of the properties of completeness and consistency for the synthesized theory. More formally, we introduce the following definitions, where \( E \) and \( E' \) denote the sets of all the available negative and positive examples, respectively.

**Definition 3 (Inconsistency)**

- A theory \( T \) is inconsistent iff \( \exists H \in T \) s.t. \( \exists N \in E : H \) is inconsistent wrt (wrt) \( N \).
- A hypothesis \( H \) is inconsistent wrt \( N \) iff \( \exists C \in H : C \) is inconsistent wrt \( N \).
- A clause \( C \) is inconsistent wrt \( N \) iff \( \exists \sigma : \)
  1. \( \text{body}(C) \sigma \subseteq \text{body}(N) \)
  2. \( \neg \text{head}(C) \sigma = \text{head}(N) \)
  3. \( \text{constraints}(C_\sigma) \sigma \subseteq \text{constraints}(N_\sigma) \)

where \( \text{body}(\varphi) \) and \( \text{head}(\varphi) \) denote the body and the head of a clause \( \varphi \), respectively. If at least one of the three conditions above is not met, we say that \( C \) is consistent wrt \( N \).

**Definition 4 (Incompleteness)**

- A theory \( T \) is incomplete iff \( \exists H \in T \) s.t. \( \exists P \in E' : H \) is incomplete wrt \( P \).
- A hypothesis is incomplete wrt \( P \) iff \( \forall C \in H : \text{not}(P \leq \sigma C) \)

Otherwise it is complete wrt \( P \).

With these definitions, we can formally define the notions of commission and omission error.

**Definition 5 (Commission/Omission error)**

Given a theory \( T \) and an example \( E \),
- \( T \) makes a commission error iff \( \exists H \in T \) s.t. \( \exists C \in H : C \) is inconsistent wrt \( E \)
- \( T \) makes an omission error iff \( \exists H \in T : H \) incomplete wrt \( E \)

When a commission error occurs, it becomes necessary to specialize the inconsistent clause \( C \) so that the new clause \( C' \) restores the consistency property of the theory. When an omission error occurs, it becomes necessary to generalize the incomplete hypothesis \( H \) so that the new hypothesis \( H' \) restores the completeness property of the theory. This points out that the process of abstract diagnosis of an incorrect theory is performed at different levels of granularity, according to the type of error found. Specifically, if a commission error occurs, the diagnosis can be carried as far as the level of a single inconsistent clause, which is the only cause of the commission error, while if an omission error occurs, we are compelled to limit the scope of the diagnosis process to the coarser level of hypotheses, that is to say, the cause of an omission error is a single hypothesis.

As to the debugging process, commission errors can be solved by exploiting properly a downward refinement operator, while, dually, upward refinement operators can cope with omission errors.

As we pointed out in [Semeraro et al., 96], in a logic framework for the inductive synthesis of Datalog theories from facts, a fundamental problem is the definition of locally finite, proper and complete (ideal) refinement operators. Indeed, when the aim is to develop incrementally a logic program, that should be correct with respect to its intended model at the end of the development process, it becomes relevant to define operators that allow a stepwise (incremental) refinement of too weak or too strong programs [Komorowski & Trcek, 94]. The ideality of the refinement operators plays a key role when the efficiency and the effectiveness of the design process is an negligible requirement. Unfortunately, when full Horn clause logic is chosen as representation language and either \( \theta \)-subsumption or implication is adopted as generalization model, there exist no ideal refinement operators [van der Laag & Nienhuys-Cheng, 94a; 94b; van der Laag, 95]. On the contrary, they do exist under the weaker, but more mechanizable and manageable ordering induced by \( \theta_{\alpha} \)-subsumption, as proved in [Esposito et al., 96; Semeraro et al., 96].

5. Upward Refinement

The upward refinement operator is inspired from the Interference Matching proposed by Hayes-Roth and McDermott (1977). Differently from the original operator, our algorithm works on clauses rather than on Parameterized Structural Representations (PSR’s). Therefore, it extends the notion of maximal abstraction to the concept of least general generalizations (lgg) under \( \theta_{\alpha} \)-subsumption.

**Definition 6 (lgg) A least general generalization under \( \theta_{\alpha} \)-subsumption of two clauses is a generalization which is not more general than any other such generalization, that is, it is either more specific than or not comparable to any other such generalization.**

Formally, given two Datalog clauses \( C_1, C_2, C \) a lgg under \( \theta_{\alpha} \)-subsumption of \( C_1, C_2 \) iff:

1. \( C \leq_{\alpha} C_i, i=1,2 \)
2. \( \forall D \text{ s.t. } C \leq_{\alpha} D, i=1,2 : \text{not}(D <_{\alpha} C) \)

\( \text{lgg}_{\theta_{\alpha}}(C_1, C_2) = \{ C \mid C \leq_{\alpha} C_i, i=1,2 \text{ and } \forall D \text{ s.t. } C \leq_{\alpha} D, i=1,2 : \text{not}(D <_{\alpha} C) \} \)

In the following, we describe an algorithm that computes the set of the least general generalizations under \( \theta_{\alpha} \)-subsumption of two any Datalog\(^\alpha \) clauses. Such an algorithm is a straightforward extension to Datalog\(^\alpha \) of a similar
algorithm given by Plotkin (1970). This extension is necessary since the space of Datalog clauses is not a lattice when ordered by \( \theta_{OI} \)-subsumption [Semeraro et al., 94], while it is a lattice when ordered by \( \theta \)-subsumption [Plotkin, 70].

Preliminarily, we extend the definition of selection in [Plotkin, 70] to Datalog\(^{OA}\) clauses.

**Definition 7 (Selection under object identity)** Let \( C_j \) and \( C_k \) be two Datalog\(^{OA}\) clauses. A selection under object identity of \( C_j \) and \( C_k \) is a pair of literals \( \langle c_i, d_j \rangle \), where \( c_i \in \text{core}(C_j) \) and \( d_j \in \text{core}(C_k) \), such that \( c_i \) and \( d_j \) have the same predicate symbol, sign, and arity.

**Algorithm (lgg\(_{OI}\) computation)**

Let \( C_j \) and \( C_k \) be two variable disjoint Datalog\(^{OA}\) clauses. The set of the least general generalizations under \( \theta_{OI} \)-subsumption of \( C_j \) and \( C_k \), denoted with \( \text{lgg}_{OI}(C_j, C_k) \), is a set of clauses where each clause \( G_{OI} \) is defined as follows.

\[
G_{OI} = G_{\text{core}} \cup G_{\text{constraints}}
\]

where

- \( G_{\text{core}} = \{ g \mid g = \text{lgg}(c_i, d_j), \langle c_i, d_j \rangle \text{ selection under object identity of } C_j \text{ and } C_k \text{ and } \varphi_k(x) \neq \varphi_j(y), k = 1, 2 \} \)

\( \text{lgg}(c_i, d_j) \) denotes the least general generalization of \( c_i \) and \( d_j \) computed with the algorithm given by Plotkin (1970).

The functions \( \varphi_k, k = 1, 2 \), are the substitutions such that \( G_{\text{core}}, \varphi_k \subseteq \text{core}(C_j), k = 1, 2 \). More precisely, if we call \( \phi \) the function \( \phi : \text{terms}(C_j) \times \text{terms}(C_k) \rightarrow \text{nvars} \cup (\text{consts}(C_j) \cap \text{const}(C_j)) \) s.t.

\[
\phi(t_i, s_j) = \begin{cases} 
  t_i & \text{if } t_i = s_j \\
  X & \text{otherwise}
\end{cases}
\]

where \( \langle c_i, d_j \rangle \) is a selection under object identity of \( C_j \) and \( C_k \), \( c_i = P(t_1, t_2, \ldots, t_m), d_j = P(s_1, s_2, \ldots, s_m) \), \text{nvars} denotes a set of new variables and \( X \) is in \( \text{nvars} \), then \( \varphi_1 \) and \( \varphi_2 \) are the projections of the inverse function of \( \phi \) onto \( \text{terms}(C_j) \) and \( \text{terms}(C_k) \), respectively.

- \( G_{\text{constraints}} = \{ x \neq y \mid \varphi_k(x) \neq \varphi_j(y), k = 1, 2 \} \)

The only difference of this algorithm with respect to Plotkin’s (1970) lies in the fact that it takes into account the (OI) axiom in order to determine a partition of the literals in \( G_{OI} \). This allows the algorithm to find the set of all the possible lgg’s under object identity rather than the unique lgg under \( \theta \)-subsumption.

**6. Downward Refinement**

Differently from the upward refinement operator, the downward refinement operator proposed in this paper is completely novel. Essentially, it relies on the addition of a non-redundant literal to a clause that turns out to be inconsistent wrt a negative example, in order to restore the consistency property of the clause. The space in which such a literal should be searched for is potentially infinite and, in any case, its size is so large that an exhaustive search is unfeasible.

We can formally define the search space as the partially ordered set (poset) \( (L/\sim_{OUI}, \leq_{OUI}) \), where \( L/\sim_{OUI} \) is the quotient set of the Datalog linked clauses and \( \leq_{OUI} \) is the quasi ordering relation defined in Section 3, which can be straightforwardly extended to equivalence classes under \( \sim_{OUI} \) [Semeraro et al., 94]. Henceforth, we will always work on the quotient set \( L/\sim_{OUI} \) and, when convenient, we will denote with the name of a clause the equivalence class it belongs to.

The novelty of the operator consists in focusing the search into the portion of the space that contains the solution of the diagnosed commission error. This peculiarity is the result of an analysis of the algebraic structure of the search space.

The search is firstly performed in the space of positive literals. This space contains information coming from the positive examples used to synthesize the current theory, but not yet exploited by it. When the search in this space fails, the algorithm autonomously performs a representation change, that allows it to extend the search to the space of the program clauses rather than the definite ones. In other words, the search is performed into a space of negative literals, built by taking into account the negative example that caused the commission error.

First of all, given a hypothesis \( H \) which is inconsistent wrt a negative example \( N \), the process of abstract diagnosis detects all the clauses of \( H \) that caused the inconsistency. Let us suppose that the subset of the positive examples \( \theta_{OUI} \)-subsumed by an inconsistent clause \( C \) is \( \{ P_j, P_{j'}, \ldots, P_n \} \). The search process aims at finding one of the most general downward refinements under object identity of \( C \) against \( N \) given \( P_j, P_{j'}, \ldots, P_n \), denoted with \( \text{mgdr}_{OUI}(C, N) \mid P_j, P_{j'}, \ldots, P_n \) and formally defined as follows.

\[
\text{mgdr}_{OUI}(C, N) \mid P_j, P_{j'}, \ldots, P_n = \{ M \in \text{mgdr}_{OUI}(C, N) \mid P_j <_{OUI} M, j = 1, 2, \ldots, n \}
\]

where the superset of the most general downward refinements under object identity of \( C \) against a negative example \( N \), denoted by \( \text{mgdr}_{OUI}(C, N) \), is formally defined as:

\[
\text{mgdr}_{OUI}(C, N) = \{ M \mid M <_{OUI} C, M \text{ consistent wrt } N, \forall D \leq_{OUI} C, D \text{ consistent wrt } N : \text{not}(M <_{OUI} D) \}
\]

Throughout this section, we shall denote with \( C \) a clause that needs to be specialized, since it is inconsistent wrt an example \( N \). More precisely, the body of \( C \) needs to be subjected to a suitable process of downward refinement in order to restore the consistency property.
Let us consider the problem of finding one of the clauses in the set \( \text{mgdr}_{\theta} (C, N | P_1, P_2, \ldots, P_n) \). Since the downward refinements we are looking for must satisfy the property of maximal generality, it may happen that the specializations of the clause \( C \) are overly general, even after some refinement steps. This suggests us the possibility of further exploiting the positive examples in order to specialize \( C \). Specifically, if there exists a literal that, when added to the body of \( C \), is able to discriminate from the negative example \( N \) that caused the inconsistency of \( C \), then the downward refinement operator should be able to find it. The resulting specialization should restore the consistency of the clause \( C \), by refining it into a clause \( C' \) which still \( \theta_{C, C'} \)-subsumes all the positive examples exploited to synthesize the theory.

The process of refining a clause by means of positive literals can be described as follows. For each \( P_i, i = 1, 2, \ldots, n \), let us suppose that there exist \( n_i \) distinct substitutions such that \( C \theta_{C, C'} \)-subsumes \( P_i \). Then, let us consider all the possible \( n \)-tuples of substitutions obtained by picking one of such substitutions for every positive example. Each of these substitutions is used to produce a distinct residual, consisting of all the literals in the positive example that are not involved in the \( \theta_{C, C'} \)-subsumption test, after having properly turned their constants into variables. Formally, a residual can be defined as follows.

**Definition 8 (Residual)** Let \( C \) be a clause, \( E \) an example, and \( \sigma_j \) a substitution s.t. \( \text{body}(C) \sigma_j \subseteq \text{body}(E) \) and \( \text{constraints}(C) \sigma_j \subseteq \text{constraints}(E) \). A residual of \( E \) wrt \( C \) under the mapping \( \sigma_j \), denoted by \( \Delta(E, C, \sigma_j) \), is the following set of literals:

\[
\Delta(E, C, \sigma_j) = \text{body}(E) \sigma_j^{-1} \setminus \text{body}(C)
\]

where \( \sigma_j^{-1} \) is the extended antisubstitution (or inductive substitution) obtained by inverting the corresponding substitution \( \sigma_j \). Indeed, an antisubstitution is a mapping from terms into variables [Siekmann, 90]. When a clause \( C \theta_{C, C'} \)-subsumes an example \( E \) through a substitution \( \sigma_j \), then it is possible to define a corresponding antisubstitution, \( \sigma_j^{-1} \), which is exactly the inverse function of \( \sigma_j \). Then, \( \sigma_j^{-1} \) maps some constants in \( E \) to variables in \( C \). Not all constants in \( E \) have a corresponding variable according to \( \sigma_j^{-1} \). Therefore, in Def. 8, we introduce the extension of \( \sigma_j^{-1} \), denoted with \( \sigma_j^{-1} \), that is defined on the whole set of constants occurring in \( E, \text{consts}(E) \), and takes values in the set of the variables of the language:

\[
\sigma_j^{-1}(c_i) = \begin{cases} 
\sigma_j^{-1}(c_i) & \text{if } c_i \in \text{vars}(C) \sigma_j \\
\_ & \text{otherwise}
\end{cases}
\]

Henceforth, variables denoted by \( _{\_} \) will be called new variables and managed as in Prolog. The residuals obtained from the positive examples \( P_i, i = 1, 2, \ldots, n \), can be exploited to build a space of complete positive downward refinements, denoted with \( P \), and formally defined as follows.

\[
P = \bigcup_{i=1,2,\ldots,n} \bigcap_{j=1,2,\ldots,n} \Delta(P_i, C)
\]

where \( \Delta(P_i, C) \) denotes one of the \( n_i \) residuals of \( P_i \) wrt \( C \) and \( \bigcap_{j=1,2,\ldots,n} \Delta(P_i, C) \), when \( j \) takes one of the values in \( \{1, 2, \ldots, n \} \), is the set of the literals common to an \( n \)-tuple of residuals (one residual for each positive example \( P_i \), \( k = 1, 2, \ldots, n \)).

Moreover, let us denote with \( \theta_{j} j = 1, 2, \ldots, m \), all the substitutions which make \( C \) inconsistent wrt \( N \). Let us define a new set of literals:

\[
S = \bigcup_{j=1,2,\ldots,m} \Delta(N, C)
\]

Then, the following proposition holds.

**Proposition 4.** Given a clause \( C \), that \( \theta_{C, C'} \)-subsumes the positive examples \( P_1, P_2, \ldots, P_n \), and is inconsistent wrt the negative example \( N \), then any linked clause \( C' = C \cup \{ l \} \), with \( l \in P \setminus S \), is in \( \text{mgdr}_{\theta} (C, N | P_1, P_2, \ldots, P_n) \).

Formally:

\[
\{ C' \mid C' = C \cup \{ l \}, l \in P \setminus S \} \subseteq \text{mgdr}_{\theta} (C, N | P_1, P_2, \ldots, P_n)
\]

Note that \( l \) is an element of \( \text{body}(C) \).

Proposition 4 states that every downward refinement built by adding a literal in \( P \setminus S \) to the inconsistent clause \( C \) restores the properties of consistency and completeness of the original hypothesis. Moreover, it is one of the most general downward refinements of \( C \) against \( N \).

When the space \( P \setminus S \) does not contain any solution to the problem of specializing an inconsistent clause, the search is automatically extended to the space of negative literals. Specifically, let us suppose that the operator did not succeed in refining a clause wrt a negative example \( N \) in a complete and consistent way. In such a case, a change of representation must be performed in order to search for literals in another space, corresponding to the quotient set of the Datalog \( \neg \)-linked clauses. Therefore, it is necessary to define a new target space, called the space of negative downward refinements. Given a clause \( C \), an example \( N \) and the set of all substitutions \( \theta_j j = 1, 2, \ldots, m \), such that \( C \) is inconsistent wrt \( N \), the space of negative downward refinements, denoted with \( S_n \), is the following set of literals:

\[
S_n = \text{neg}(S) = \text{neg}(\bigcup_{j=1,2,\ldots,m} \Delta(N, C))
\]
where, given a set of literals \( \varphi = \{ l_1, l_2, ..., l_n \} \), \( n \geq 1 \), \( \text{neg}(\varphi) \) denotes the set of literals \( \{ \neg l_1, \neg l_2, ..., \neg l_n \} \).

As for the process of downward refinement by positive literals, we are interested in a specific subset of \( \mathbb{S} \), because of the properties satisfied by its elements. Such a subset, called space of consistent negative downward refinements, is denoted with \( \mathbb{S} \) and is defined as follows.

\[
\mathbb{S} = \text{neg}(\cap_{i=1,2,...,m} \Delta(N, C))
\]

The reason why the operator focuses onto the subset \( \mathbb{S} \), rather than onto the whole set \( \mathbb{S} \), lies in the following result:

**Proposition 5.** Given a clause \( C \), an example \( N \) and the set of all substitutions \( \theta, j=1,2,...,m \), such that \( C \) is inconsistent wrt \( N \), then any linked program clause \( C' = C \cup \{ l \} \), with \( l \in \mathbb{S} \), is in \( \text{mgdr}_{/\theta}(C, N) \).

**Formally:**

\[
\{ C' \mid C' = C \cup \{ l \}, l \in \mathbb{S} \} \subseteq \text{mgdr}_{/\theta}(C, N)
\]

Note that \( l \) is a negated literal occurring in the body of \( C' \); thus \( C' \) is a Datalog\(^{-}\) clause.

Proposition 5 easily extends to any linked literal \( l \) which introduces new variables, due to negation-as-failure rule. Generally speaking, we can say that, given a clause \( C \) and an example \( N \) such that \( C \) is inconsistent wrt \( N \) due to some substitutions \( \theta, j=1,2,...,k \), the search for a complete and consistent hypothesis can be viewed as a two-stage process: the former stage searches into the space \( \mathbb{P} - \mathbb{S} \), the latter into \( \mathbb{S} \). By means of Propositions 4 and 5, we are now able to formally define our novel downward refinement operator on the space of constant-free Datalog linked program clauses, denoted with \( \rho^{\text{wmr}}_{/\theta} \).

**Definition 9** (\( \rho^{\text{wmr}}_{/\theta} \))

\[
\rho^{\text{wmr}}_{/\theta} : \mathbb{L} \rightarrow 2^\mathbb{L}, \forall C \in \mathbb{L}: \rho^{\text{wmr}}_{/\theta}(C) = \{ C' \mid C' = C \cup \{ l \}, l \in (\mathbb{P} - \mathbb{S}) \cup \mathbb{S} \}
\]

**Proposition 6.** The downward refinement operator \( \rho^{\text{wmr}}_{/\theta} \) is ideal.

The ideality of the refinement operator \( \rho^{\text{wmr}}_{/\theta} \) is owed to the peculiar structure of the search space when ordered by the relation \( \leq_{/\theta} \). In the same search space ordered by \( \theta \)-subsumption, an ideal refinement operator does not exist [van der Laag & Nienhuys-Cheng, 94b].

### 7. Application to Document Understanding

We implemented the operators described in Sections 5 and 6 in an incremental system, called INCR/H, in order to compare the performance of Datalog theories synthesized incrementally to those synthesized in batch mode. Several experiments were carried out in the area of electronic document understanding. For all the experiments, we considered a database of 30 single-page A4 documents (Olivetti letters), in each of which we labelled the layout components with the corresponding logical meaning (logotype, signature, body, reference number, sender, date, receiver) in order to obtain the clauses representing the examples. Each labelled object is a positive example for the component it is associated to and, at the same time, is a negative example for all the other components.

The experiments have been replicated ten times, by randomly splitting the database into two subsets, namely a learning set (with size 20) and a test set (with size 10). In turn, the learning set has been subdivided into training set and tuning set, with size 5 and 15, respectively, and has been exploited in two distinct ways, according to the mode - batch or incremental - adopted to synthesize the Datalog theories. In the former case, this set has been entirely given to the batch system INDUBI/H [Esposito et al., 94] as its input. In the latter case, only the training set has been used by INDUBI/H in order to produce the set of hypotheses representing the first version of the theory; then, the tuning set has been exploited to correct incrementally both omission and commission errors, if any, through the refinement process described in Sections 5 and 6. Lastly, the test set has been exploited to evaluate the error rate of the inferred theories on unclassified components.

Table 1 reports the information concerning the experimental setup, i.e. the mean number of positive and negative examples used by the different phases. Figure 1 shows the results of the comparison between the theory synthesized in batch mode and that obtained incrementally, as concerns the error rate on the test set and the computational time taken by the system to produce the theories. Specifically, the batch time refers to the training set for the batch mode, while the incremental time is computed as the sum of the computational time concerning the training set for the incremental mode plus the time concerning the tuning set. Values concerning the error rate are percentages, while those concerning the time are expressed in seconds. All the reported figures refer to the average on the ten replications.

<table>
<thead>
<tr>
<th></th>
<th>Learning set (batch)</th>
<th>Training set (incr.)</th>
<th>Tuning set</th>
<th>Test set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#pos + #neg</td>
<td>#pos + #neg</td>
<td>#pos + #neg</td>
<td>#pos + #neg</td>
</tr>
<tr>
<td>logotype</td>
<td>20+220</td>
<td>5+55</td>
<td>15+165</td>
<td>10+110</td>
</tr>
<tr>
<td>signature</td>
<td>19+221</td>
<td>4+56</td>
<td>15+165</td>
<td>10+110</td>
</tr>
<tr>
<td>body</td>
<td>20+220</td>
<td>5+55</td>
<td>15+165</td>
<td>10+110</td>
</tr>
<tr>
<td>ref</td>
<td>31+209</td>
<td>8+52</td>
<td>23+157</td>
<td>15+105</td>
</tr>
<tr>
<td>sender</td>
<td>23+217</td>
<td>6+54</td>
<td>17+163</td>
<td>12+108</td>
</tr>
<tr>
<td>date</td>
<td>24+216</td>
<td>6+54</td>
<td>18+162</td>
<td>12+108</td>
</tr>
<tr>
<td>receiver</td>
<td>24+216</td>
<td>6+54</td>
<td>18+162</td>
<td>12+108</td>
</tr>
</tbody>
</table>

**Table 1.** Sizes of the example sets in the experimentations concerning the problem of document understanding.
Table 2. Statistical results.

<table>
<thead>
<tr>
<th>Document understanding</th>
<th>t test</th>
<th>Wilcoxon test</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>batch vs. incremental</td>
<td>batch vs. incremental</td>
</tr>
<tr>
<td>Error rate</td>
<td>Time</td>
<td>W value</td>
</tr>
<tr>
<td>t value</td>
<td>sign. value</td>
<td>t value</td>
</tr>
<tr>
<td>logo</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>signature</td>
<td>1.3</td>
<td>.2259</td>
</tr>
<tr>
<td>body</td>
<td>1.353</td>
<td>.2091</td>
</tr>
<tr>
<td>ref</td>
<td>.318</td>
<td>.7577</td>
</tr>
<tr>
<td>sender</td>
<td>2.333</td>
<td>.0445</td>
</tr>
<tr>
<td>date</td>
<td>-1.849</td>
<td>.0975</td>
</tr>
<tr>
<td>receiver</td>
<td>1.769</td>
<td>.1108</td>
</tr>
</tbody>
</table>

Table 2 illustrates the results of two statistical methods exploited to evaluate the significance of the observed differences as to the error rate and time for each type of logical label, namely the paired t test (parametric) and the Wilcoxon test (non parametric). For a thorough explanation of the two statistical tests, refer to [Orkin & Drogin, 90]. The t test has been performed as two-sided test at a 0.01 level of significance, while the Wilcoxon test both at 0.05 and at 0.01 level.

Each entry in the table contains the t value and the corresponding significance value for the t test, the W value and the corresponding critical value, along with the sample sizes, for the Wilcoxon test.

It is well-known that the t test requires that the population data be normally distributed, when used with small sample sizes (less than 30). Conversely, the Wilcoxon test does not make any assumption on the distribution of the population data. In our setting, the sample size is 10, i.e. the number of replications, thus the t test might seem to be unsuitable. However, we performed preventively a normality test in order to establish whether the population data are normally distributed. Such a test allows us to state that the population is normally distributed at a 0.01 level of significance.

Figure 1 shows that the batch theories outperform the incremental ones for all the classes, with the exception of sender and date, as regards the error rate; on the contrary, as to the computational times the incremental system outperforms the batch one in all cases. However, the t test reveals no statistically significant difference between the error rates (clear boxes in Table 2), but a great difference in the case of computational time (shaded boxes). According to the Wilcoxon test the results are the same.

8. Conclusions and Future Work

In this paper, we presented a logic framework for the incremental inductive synthesis of Datalog theories. It allows us to define algorithms for abstract diagnosis and debugging of incorrect theories, which strongly rely on effective and efficient (ideal) refinement operators. These algorithms have been implemented in a new version of an incremental system for inductive synthesis of Datalog theories, called INCR/H, whose performance has been successfully tested on the real-world problem of electronic document understanding.

Future work will aim at integrating INCR/H in the learning server component of a prototypical intelligent digital library service, called IDL. In IDL, document classification and understanding play a key role in the process of information capture and semantic indexing of the stored documents [Esposito et al., 97].
References


Appendix

Proof (Proposition 1)
The difference between regarding the same clause in Datalog or in Datalog\textsuperscript{\textalpha} is that, in the first case, it is interpreted without any constraint on the variables appearing in it, while, in the second case, additional literals expressing the OI-constraints are implicitly assumed. If we progressively eliminate from the clause all the possible combinations of inequations and introduce new clauses reflecting their unification, every possible substitution we can think of will involve one of the combinations already considered (maybe the empty one), so we can refer to the corresponding clause which we introduced and which eliminated the noisy constraints.

These observations lead to the following algorithm, that, given a Datalog clause \( C \), computes the set of Datalog\textsuperscript{\textalpha} clauses \( C' = \{ C_1, C_2, \ldots, C_n \} \) such that \( T_c \uparrow \omega = T_{c'} \uparrow \omega \).

**Algorithm 1**

function \( \text{GenerateDatalogOIClauses} \left( C : \text{DatalogClause} \right) : \text{DatalogOIClauses} \)

begin
\( C' := \{ \} \);
for \( k = 0, 1, \ldots, \left| \text{vars}(C) \right| \) do

foreach combination of \( k \) variables out of \( \left| \text{vars}(C) \right| \) do

begin
Define some ordering between the \( k \) variables;

foreach permutation with replacement of \( k \) constants out of \( \left| \text{consts}(C) \right| \) do

for \( h = 1, 2, \ldots, \left| \text{vars}(C) \right| - k \) do

begin
\( C_h' := \{ \} \);

foreach combination \( (V_i)_{i=1,2,\ldots,h} \) of the remaining variables s.t. \( \forall i, j = 1, 2, \ldots, h, i < j : |V_i| = r_i, r_i \neq r_j \) do

begin
Build a clause \( D \) by replacing the \( l \)-th \( (l = 1, 2, \ldots, k) \) variable of the combination with the \( l \)-th constant of the permutation and, \( \forall i = 1, 2, \ldots, h \) all the \( r_i \) variables belonging to \( V_i \) with one new variable;

if \( \forall i, j = 1, 2, \ldots, k, i \neq j; r_i \neq r_j \) then Insert \( D \) in \( C_h' \);

elsif there exists no renaming of \( D \) in \( C_h' \) another clause then Insert \( D \) in \( C_h' \);

end;

end;

\( C' := C' \cup C_h' \);

end;

end;

return \( C' \)

end.

By definition, the \( T_p \) operator acts only on the ground instances of the clauses of the program to which it is applied. As a consequence, proving the thesis is equivalent to demonstrate that the following sets \( A \) and \( B \) are equal.

\[
A = \text{ground}(C) \quad B = \{ \text{ground}(D) \mid D \in C' \}
\]

where \( \text{ground}(E) \) denotes the set of the ground instances of the clause \( E \).

As the clauses in \( C' \) have been obtained from \( C \) by simply unifying variables (among them or with a constant), their structure is surely equal to that of \( C \). But then, when comparing an element of \( \text{ground}(C) \) with one of \( \text{ground}(C') \), it suffices to consider ground terms, which will be constants (since we work in Datalog and Datalog\textsuperscript{\textalpha}). After this introduction, let us prove the thesis (by double inclusion).

**\((A \subseteq B)\)** Any ground instance of \( C \) is obtained by substituting all the variables in \( C \) with constants.

i) If all these constants are already in \( C \), then Algorithm 1 generated (in the last step of the loop) all the possible combinations of substitutions of variables in \( C \) with constants in \( C \);

ii) If none of these constants was already in \( C \), then we will have a partition of \( \text{vars}(C), (V_i)_{i=1,2,\ldots,k} (1 \leq k \leq \left| \text{vars}(C) \right|) \), s.t. \( \forall i=1, 2, \ldots, k \) all variables in \( V_i \) were associated to the same constant \( a_i \) \( (a_i \neq a_j \ \forall i \neq j) \). But then Algorithm 1 generated (in the first step of the loop) a clause which unified among them all the variables in each \( V_i \), to which we can apply the same substitution, as the noisy OI-constraint has been removed;

iii) If some of these constants were already in \( C \) and others were not, Algorithm 1 applied the same substitutions on the constants which were already present, and unified the remaining variables according to the partition introduced by the remaining constants (see ii), thus generating a clause to which it is possible to apply the
same substitution.
Please, note that i) and ii) can be seen as particular cases of iii).

(B ⊆ A) On the other hand, given any ground instance of a clause \( C_i \in C' \), it can be obtained immediately from \( C \) by substituting every variable with the constant which appears in the same position in that ground instance of \( C_i \), because in \( C \) we do not have the OI-constraint. Indeed, the algorithm obtained \( C' \) from \( C \) by leaving untouched all the constants, and at most by unifying some variables.

\[ (B \subseteq A) \]

Proof (Corollary 1)
Even for this result, the proof is constructive. It suffices applying Algorithm 1 in the proof of Prop. 1 to every clause in \( P \), and taking as \( P' \) the set of all the clauses generated by the following algorithm:

Algorithm 2
function GenerateDatalogOIProgram ( \( P \): DatalogProgram ) : DatalogOIProgram
begin
    \( P' := \{ \} \);
    foreach clause \( C \in P \) do
        \( C' := \text{GenerateDatalogOIClauses} (C) \);
        \( P' := P' \cup C' \);
    end;
    return \( P' \);
end;

Again, the semantic equivalence is cast (by definition of \( T_p \)) to proving the equality of the following sets:
\[
A = \{ \text{ground}(D) \mid D \in P \} \\
B = \{ \text{ground}(D) \mid D \in P' \}
\]

(A ≤ B) Given a ground instance of a clause \( C \in P \), \( C \) has been surely considered in some step of Algorithm 2, thus it generated a set of clauses \( C' \subseteq P' \). But then, the proof of \((A \subseteq B)\) in Prop. 1 holds for the (unique) ground instance of \( C \) and for the set of ground instances of clauses in \( C' \).

(B ⊆ A) Given a ground instance of a clause \( C' \in P' \), \( C' \) was generated by Algorithm 2 in correspondence of a clause \( C \in P \). But then, the proof of \((B \subseteq A)\) in Prop. 1 holds for the ground instance of \( C' \) and for the set of ground instances of \( C \).

q.e.d.

Proof (Proposition 2)
a) \( C \subseteq C \) implies \( C_{\alpha} \{ \} \subseteq C_{\alpha} \), thus \( C \leq_{\alpha} C \).
b) If \( C \leq_{\alpha} D \) and \( D \leq_{\alpha} E \) then there exist the substitutions \( \sigma \) and \( \theta \) such that \( D_{\alpha} \sigma \subseteq C_{\alpha} \) and \( E_{\alpha} \theta \subseteq D_{\alpha} \). Therefore, it holds \( E_{\alpha} \theta \sigma \subseteq D_{\alpha} \sigma \subseteq C_{\alpha} \), where \( \theta \sigma \) denotes the composite of the substitutions \( \theta \) and \( \sigma \). This proves \( C \leq_{\alpha} E \).

q.e.d.

Proof (Proposition 3)
\[ \Rightarrow \] From Definition 2, \( C \leq_{\alpha} D \) means that there exists a substitution \( \sigma \) s.t. \( D_{\alpha} \sigma \subseteq C_{\alpha} \) implies
\[
(\text{core}(D_{\alpha}) \cup \text{constraints}(D_{\alpha})).\sigma \subseteq \text{core}(C_{\alpha}) \cup \text{constraints}(C_{\alpha}) \\
(\text{core}(D_{\alpha}).\sigma \cup \text{constraints}(D_{\alpha})).\sigma \subseteq \text{core}(C_{\alpha}) \cup \text{constraints}(C_{\alpha})
\]

But inequalities cannot occur in a Datalog clause, thus it yields straightforwardly:
\[
\text{core}(D_{\alpha}).\sigma \cap \text{constraints}(D_{\alpha}).\sigma = \emptyset \\
\text{core}(C_{\alpha}) \cap \text{constraints}(C_{\alpha}) = \emptyset \\
\text{core}(D_{\alpha}).\sigma \cap \text{constraints}(C_{\alpha}) = \emptyset \\
\text{constraints}(D_{\alpha}).\sigma \cap \text{core}(C_{\alpha}) = \emptyset
\]

By simple algebraic manipulations, it is possible to obtain:
\[
\text{core}(D_{\alpha}).\sigma \subseteq \text{core}(C_{\alpha}) \text{ and} \\
\text{constraints}(D_{\alpha}).\sigma \subseteq \text{constraints}(C_{\alpha})
\]

\[ \Leftarrow \] Trivial.

q.e.d.
Proof (Proposition 4)
In order to prove $C' \prec_{\omega_l} C$, let us observe that in the space $(L_{\prec_{\omega_l}}, \preceq_{\omega_l})$ the set of all constant-free upward refinements of a clause $C'$ corresponds to the set $2^{C'}$, thus each proper upward refinement of $C'$ has a number of literals less than the number of literals of $C'$ [VanLehn, 89]. Therefore $C \subseteq C' \Rightarrow C' \prec_{\omega_l} C$, i.e. $C'$ is a proper downward refinement of $C$ under $\theta_{\omega_l}$-subsumption.

Let us show now that $C'$ is consistent wrt $N$. First of all, observe that
\[
\forall j = 1, 2, ..., m : \neg \text{head}(C). \theta_j = \neg \text{head}(C). \theta_j = \text{head}(N).
\]
Moreover,
\[
\forall j = 1, 2, ..., m : \text{body}(C). \theta_j = \text{body}(C). \theta_j = \text{body}(C). \cup \{ l \}. \theta_j = \text{body}(C). \theta_j \cup \{ l \}. \theta_j
\] (1)
\[
l \in P - S \Rightarrow l \not\in S = \cup_{j=1}^{l} \Delta(N, C) \Rightarrow \forall j = 1, 2, ..., m : l \not\in \Delta(N, C).
\]
By definition of $P$, $l \in P \Rightarrow l \in C$, then $\forall j = 1, 2, ..., m : l \not\in \text{body}(N). \theta_j = l \not\in \text{body}(N) \Rightarrow \{ l \}. \theta_j \not\in \text{body}(N).
\]
Then, looking back at (1), we can conclude that: $\forall j = 1, 2, ..., m : \text{body}(C). \theta_j \not\in \text{body}(N).
\]
This proves that $C'$ is consistent wrt $N$. Indeed, any other substitution causing the inconsistency of $C'$ would be a superset of a $\theta_j$ $(j=1, 2, ..., m)$ because of our assumption that they were the only possible substitutions under object identity s.t. $\text{body}(C). \theta_j \subseteq \text{body}(N)$ and we have just proved that each of them makes $C'$ consistent wrt $N$.

Now suppose that $\exists F$ which is consistent wrt $N$ and s.t. $C' \prec_{\omega_l} F \leq_{\omega_l} C = |C| + 1 > |F| \geq |C| \Rightarrow |F| = |C|.
\]
But $F$ is a specialization of $C$, then it can be inferred that $F \sim_{\omega_l} C$. Thus $F$ is inconsistent wrt $N$, just as $C$.

According to the hypotheses of the proposition:
\[
\forall k = 1, 2, ..., n : \text{head}(C). \sigma_k = \text{head}(P). \sigma_k \subseteq \text{body}(P). \sigma_k = \text{body}(P). \sigma_k \subseteq \text{body}(P).
\]
Then, body($C). \sigma_k = \text{body}(C). \sigma_k \cup \{ l \}. \sigma_k \subseteq \text{body}(P). \forall k = 1, 2, ..., n.\]
Then, $P_k \leq_{\omega_l} C'$.
\]
q.e.d.

Proof (Proposition 5)
As to the proof that $C'$ is a proper downward refinement of $C$ under $\theta_{\omega_l}$-subsumption, refer to the previous proof.

Given a program clause $C' = C \cup \{ l \}$, with $l \in S$, in order to prove that $C'$ is consistent wrt $N$, let us suppose (reductio ad absurdum) there exists a substitution $\sigma_k$ s.t. $C'$ is inconsistent wrt $N$. Then, from Def. 3, it results that:
\[
1) \text{body}(C). \sigma_k \not\subseteq \text{body}(N) 2) \neg \text{head}(C). \sigma_k = \text{head}(N) 3) \text{constraints}(C). \sigma_k \subseteq \text{constraints}(N).
\]
As a consequence, $\sigma_k$ is also one of the $k$ substitutions that make $C$ inconsistent wrt $N$. We also have from the hypotheses of the proposition:
\[
\text{body}(C). \sigma_j = \text{body}(C). \sigma_j = \text{body}(C). \sigma_j = \text{body}(C). \sigma_j = \text{body}(C). \sigma_j = \text{body}(C). \sigma_j = \text{body}(C). \sigma_j = \text{body}(C). \sigma_j \subseteq \text{body}(N).
\]
with $l \in S = \text{neg}(\cap_{j=1}^{l} \Delta(N, C))$. But:
\[
\{ l \}. \sigma_k \subseteq S, \sigma_k = \text{neg}(\Delta(N, C)), \sigma_k = \text{neg}(\text{body}(N), \sigma_k \not\subseteq \text{body}(C). \sigma_k) = \text{neg}(\text{body}(N) \not\subseteq \text{body}(C). \sigma_k) \not\subseteq \text{body}(N) \not\subseteq \text{body}(N).
\]
and:
\[
\{ l \}. \sigma_k \subseteq \text{body}(C). \sigma_k \subseteq \text{body}(C). \sigma_k \not\subseteq \text{body}(N), \text{according to } l.
\]
But this is impossible since body(N) \cap neg(body(N)) = \emptyset.

In order to prove that $C'$ is in $mgd_{\omega_l}(C, N)$, it remains to demonstrate that:
\[
\forall D, D \preceq_{\omega_l} C, D \text{ consistent wrt } N, \text{ not}(C' \prec_{\omega_l} D).
\]

Suppose (ad absurdum) that there exists $D$ s.t. $D \preceq_{\omega_l} C, D \text{ consistent wrt } N \text{ and } C' \prec_{\omega_l} D$.

Therefore, from (2) and (3), it results: $|D| \geq |C|.
\]
By hypothesis, C is a generalization of D, but the only constant-free upward refinement of D having the same number of literals of D is D itself. Thus, C = D and this is a contradiction because C is inconsistent wrt N, whilst D is consistent wrt N by hypothesis.
\]
q.e.d.

Proof (Proposition 6)
(properness)
\[
\rho^{\text{cons}}_{\omega_l} \text{ is proper as a consequence of the definition of } mgd_{\omega_l}(C, N) \text{ and of Propositions 4 and 5. Indeed, }
\]
\[
\rho^{\text{cons}}_{\omega_l}(C) \subseteq mgd_{\omega_l}(C, N).
\]
(local finiteness)
The choice of $l$ in $\rho^{\text{cons}}_{\omega_l}$ is related to the construction of the sets $S, P$ and $S$. Note that the number of substitutions s.t. a clause $C \theta_{\omega_l}$-subsumes a clause $D$ is finite and equal to:
\[
|\text{vars}(D)| \times (|\text{vars}(D)| - 1) \times \ldots \times (|\text{vars}(D)| - |\text{vars}(C)| - 1).
\]
13
It is worthwhile to note that $S_c$ is an intersection of a finite number of residuals, by definition. This number depends on the (finite) number of substitutions between the clause $C$ to be refined and the example $N$ which causes the problem of inconsistency. In turn, each residual is a finite difference-set of literals between two clauses. Thus, $S_c$ is finite and computable. $P$ is also an intersection of a finite number of difference-sets between two clauses. This number depends on the number of substitutions between $C$ and the positive examples already processed. Finally, the set $S$ is the union of a finite number of difference-sets between two clauses. This number depends on the number of substitutions between $C$ and $N$. Since these sets are both finite and computable, $\rho^\text{cons}_{oi}$ is locally finite.

(completeness)

Let $C, D$ be two clauses such that $D <_{oi} C (C, D \in L)$. In this case, there exist some substitutions $\sigma_j, j=1, 2, \ldots, s$ s.t.

$|\Delta(D, C)| = r$.

For a given $j \in \{1, 2, \ldots, s\}$, let us consider the literals in $\Delta(D, C)$. Then, we may write $D$ as follows:

$D = C.\sigma_j \cup \{l_1, l_2, \ldots, l_s\}$, where $l_i \sigma_j^{-1} \in \Delta_j(D, C), k = 1, 2, \ldots, r$.

We can build the following set of clauses:

$\{F_h\}_{h=0, 1, \ldots, r}$, where $F_h = C.\sigma_j \cup \{l_1, l_2, \ldots, l_h\}$, for $h = 0, 1, \ldots, r$.

Note that: $F_0 = C.\sigma_j$ and $F_r = D$.

In order to demonstrate the completeness property, it is to be proven that:

$\forall k = 0, 1, \ldots, r-1: F_{k+1} \in \rho^\text{cons}_{oi}(F_k)$.

For a given $k \in \{0, 1, \ldots, r-1\}$, let us consider $F_{k+1} = C.\sigma_j \cup \{l_1, l_2, \ldots, l_{k+1}\} = F_k \cup \{l_{k+1}\}$.

Let us suppose now, without loss of generality, that the database of the available positive examples is made up of the set $\{P_1, P_2, \ldots, P_n\}$ and that $N_k$ is the negative example which calls for the downward refinement operator $\rho^\text{cons}_{oi}$.

If $l_{k+1}$ is a positive literal in the body of $F_{k+1}$, then, by looking back at the definition of $\rho^\text{cons}_{oi}$, we note that we are able to build the sets $P$ and $S$ s.t. $l_{k+1} \in P - S$, and then $F_{k+1} = F_k \cup \{l_{k+1}\} \in \rho^\text{cons}_{oi}(F_k)$. In fact, the set $P$ depends on the positive examples $\{P_1, P_2, \ldots, P_n\}$, and the set $S$ depends on $N_k$, which can be chosen in such a way that $l_{k+1} \in \Delta(N_k, F_k)$ for each substitution $\gamma$, between $N_k$ and $F_k$ causing $F_k$ to be inconsistent wrt $N_k$.

If $l_{k+1}$ is a negative literal in the body of $F_{k+1}$, then by definition of $\rho^\text{cons}_{oi}$ we are able to build the set $S_c$, which in turn depends on $N_k$ and $F_k$, in such a way that $l_{k+1} \in \neg(\Delta(N_k, F_k))$ for each substitution $\gamma$ above.

q.e.d.