Intoducing Taxonomic Reasoning in LOGIDATA$^+$

Alessandro Artale and Francesca Cesarini and Giovanni Soda

Dipartimento di Sistemi e Informatica - Università di Firenze - Italia

Abstract. In this paper we give a syntax and define a denotational semantics for describing classes in LOGIDATA*, and give a subsumption algorithm that is sound, complete, and polynomial. The syntax allows us to describe both primitive and defined acyclic classes by means of the tuple and set type constructors; in particular, it is also possible to state cardinality constraints on sets. Semantics is defined by means of an interpretation function that maps the descriptions given at an intensional level into the value domain. This interpretation also takes the undefined element nil into account, thus allowing us to deal with a form of incomplete knowledge in a way that is semantically correct. After introducing defined classes and the notion of interpretation function, it is possible to formally define the concept of subsumption, an inference technique that makes it possible reasoning about objects and classes on the basis of their descriptions.

1 Outline of the Paper

This paper deals with introducing in LOGIDATA$^+$ the paradigm of taxonomic reasoning; starting from the model described in [2], a LOGIDATA* syntax by means of which is possible to describe primitive and defined acyclic classes is defined in section 2, this syntax provides for tuple and set type constructors and cardinality constraints on sets can be stated. A denotational semantics that supports a kind of incomplete knowledge about objects is described in section 3. For this purpose the functions associating values to tuple labels and object identifiers are suitalbly extended in order to take a null value into account; furthemore, the interpretation function is explicitly linked to the class definition syntax in order to obtain legal instances. Some examples of deductions on the extensional level of the Knowledge-Base are illustrated in section 4. Section 5 is devoted to characterize the notions of schema coherence and subsumpition, and the syntactic constraints for coherence are illustrated in section 6. Section 7 discusses the computational complexity of the subsumption algorithm. We make our concluding remarks in section 8. The Appendix A shows an example of schema definition and interpretation, while in the Appendix B the soundness and completeness of the subsumption algorithm are proven.

2 Schema Definition Language

A LOGIDATA* schema [2] can be defined by means of the syntax illustrated below. We note that we don't take any recursive definitions into account, in accordance with the approach usually used by the Knowledge-Representation-Models we refer to [11]. As far as <basic-type> is concerned, the actual base types depend on the implementation used; anyway, we deal with base types that refer to a countable set of values. For maintaining the class hierarchy we introduce the universal class $\top$ (the *Top* class) such that, for each object o present in the Knowledge-Base, we can say that o is a member of the universal class $\top$. Furthermore, the type associated with the <type-id> appearing in the <class-declaration> must be a tuple type. The ISA clause allows the system to exploit the mechanism of inheritance; multiple inheritance is allowed, with some restrictions on the legal schema declarations assuring type compatibility (see sect. 6).

<declaration> := <type-declaration> | <class-declaration>
<type-declaration> := **type** <type-id> = <type-constructor>
<class-declaration> := $\top$ |
class <identifier> <prim-def>
ISA <class-id>*: <type-id> |
class <identifier> <prim-def>
ISA <class-id>* <tuple-type>
<type-constructor> := <tuple-type> | <set-type>
<type> := <type-constructor> | <class-id> |
<type-id> | <basic-type>
<tuple-type> := [<component>*]
<component> := <label>:<type>
<set-type> := {<type>}$_{min, max}$ | *EmptySet*
<basic-type> := string | integer
<prim-def> := $\dot{\leq}$ | $\doteq$

This syntax allows for instance to describe the concept of father as the set of all "individuals who are persons with at least one child and all children are persons":

$$\textbf{class}\ \textit{Father} \doteq \textit{ISA Person}\ [\textit{child:Person}_{1,\infty}]$$

We show a more detailed example of schema definition in the Appendix A.

3 A Semantics Dealing with Incomplete Knowledge

We design our database system LOGIDATA* in such a way that it supports incomplete information about objects [4]. Thereafter, information about an object is added incrementally, allowing us to create an object about which nothing is known except that it belongs to the universal class $\top$ or to assert that it is an instance of some classes without necessarily specifying values for the fillers of its properties. For this purpose we redefine, with respect to [2], the v_t and δ functions as total mapping such that:

$$v_t : \mathbf{L} \longrightarrow \mathcal{V} \cup \{nil\}$$

$$\delta : \mathcal{O} \longrightarrow \mathcal{V}_{\mathrm{T}} \ \cup \ \{nil\}$$

assuming that:

- $\delta(o) =$ nil when an object is created but is not assigned to any class because nothing is known about its properties (in this case o is a member of the universal class $\top$);
- $v_t\,(l) =$ nil for labels with undefined value.

The nil element makes it possible to treat incomplete knowledge maintaining v_t and δ total mapping.

Now we associate a semantics with the LOGIDATA* syntax by means of an interpretation function that extends the definition given in [2], accounting for incomplete knowledge and for the syntactic class structure.
Let $\Phi \ : \mathbf{C} \rightarrow 2^{\mathbf{C}_P}$ be a function that associates to each class $\mathrm{C} \in \mathbf{C} = \mathbf{C}_D \cup \mathbf{C}_P$ the set of all its primitive superclasses:

$$\Phi(C) \ = \ \{C_p \in \mathbf{C}_P \ \mid \ C \mathrm{ISA} C_p \ \textit{and} \ C_p \neq C\}$$

then we have the following definition.

Definition 1 (Interpretation). *Given a LOGIDATA* schema* $\mathbf{S}$*, let* $\mathcal{T}$ *be the set of type descriptors in* $\mathbf{S}$*. Given a domain* $\Sigma = (\mathcal{O}, \delta)$*, let* $\mathcal{I}$ *be a function defined from type descriptors in* $\mathbf{S}$ *to* $2^{\mathcal{V}}$ *(i.e.,* $\mathcal{I} : \mathcal{T} \rightarrow 2^{\mathcal{V}}$*);* $\mathcal{I}$ *is a* legal interpretation function *(or, briefly,* interpretation function*) of* $\mathbf{S}$ *over* $\mathcal{V}$ *if and only if:*

1. $\mathcal{I}\,(<\textit{basic-type}>_i) = \mathcal{D}_i$
2. $\mathcal{I}\,(<\textit{tuple-type}>) = \mathcal{I}\,([l_1 : t_1 \ldots l_k : t_k]) \ =$
   $\{v_t \in \mathcal{V}_{\mathrm{T}} \ \mid \ v_t \ \textit{is defined at least on} \ \{l_1 \ldots l_k\} \ \textit{and}$
   $v_t\,(l_i) \in \mathcal{I}\,(t_i), i = 1, \ldots, k\}$
3. $\mathcal{I}\,(<\textit{set-type}>) \ = \ \mathcal{I}\,(\{t\}_{m,n}) \ = \ \{v_s \in \mathcal{V}_{\mathrm{S}} \ \mid \ m \leq \| \ v_s \subseteq \mathcal{I}\,(t) \ \| \leq n\}$*, and:*
   (a) $\mathcal{I}\,(\{t\}_{0,\infty}) \ = \ \{v_s \in \mathcal{V}_{\mathrm{S}} \ \mid \ v_s \subseteq \mathcal{I}\,(t)\}$
   (b) $\mathcal{I}\,(\textit{EmptySet}) = \{\emptyset\}$
4. $\mathcal{I}\,(<\textit{type-name}>) \ = \ \mathcal{I}\,(\mathbf{type} \ \textit{type-id} = <\textit{type}>) \ = \ \mathcal{I}\,(<\textit{type}>)$
5. $\mathcal{I}\,(<\textit{class-name}>) \ = \ \mathcal{I}\,(C) \subseteq \mathcal{O}$*, and:*
   (a) $\mathcal{I}\,(\top) \ \equiv \ \mathcal{O}$
   (b) $\mathcal{I}\,(\mathbf{class} \ C_d \doteq \mathrm{ISA} \ C_1 \ldots C_n \ \ <\textit{tuple-type}>) \ =$
   $= \ \mathcal{I}\,(C_d) \ = \ \bigcap_{i=1}^{n} \mathcal{I}\,(C_i) \bigcap \{o \in \mathcal{O} \ \mid \ \delta(o) \in \mathcal{I}\,(\mathrm{TYP}(C_d))\}$
   (c) $\mathcal{I}\,\big(\mathbf{class} \ C_p \dot{\leq} \mathrm{ISA} \ C_1 \ldots C_n \ \ <\textit{tuple-type}>\big) \ =$
   $= \ \mathcal{I}\,(C_p) \ \subseteq \ \bigcap_{i=1}^{n} \mathcal{I}\,(C_i) \bigcap \{o \in \mathcal{O} \ \mid \ \delta(o) \in \mathcal{I}\,(\mathrm{TYP}(C_p))\}$

For the interpretation of TYP(C) we have the following recursive definition.

Definition 2 (Interpretation of TYP(C)). *Given the class declarations:*
class C_0 <*prim-def*> <*tuple-type*>$_0$
class C_1 <*prim-def*> ISA C_0 <*tuple-type*>$_1$
. . . .
class C_{n+1} <*prim-def*> ISA $C_0 \ldots C_n$ <*tuple-type*>$_{n+1}$
we have
$\mathcal{I}(\mathrm{TYP}(C_0)) \quad = \mathcal{I}(<\textit{tuple-type}>_0)$
$\mathcal{I}(\mathrm{TYP}(C_1)) \quad = \mathcal{I}(\mathrm{TYP}(C_0)) \bigcap \mathcal{I}(<\textit{tuple-type}>_1)$
. . . .
$\mathcal{I}(\mathrm{TYP}(C_{n+1})) = \bigcap_{i=1}^{n} \mathcal{I}(\mathrm{TYP}(C_i)) \bigcap \mathcal{I}(<\textit{tuple-type}>_{n+1})$

This semantics, allows us to consider the ISA relationship as an inclusion between classes. Moreover, each value can have more than one type [5]: when a value is of type t, then it is of type t', too, in the case that $\mathcal{I}(t) \subseteq \mathcal{I}(t')$. In the Appendix A we show an example of interpretation.

4 Deductions on Instances

The above introduced semantics provides the LOGIDATA* database with a powerful means for manipulating the explicit information about the domain (extensional level) and thus allows the system to have an active role in deducing new relationships rather than being just a passive repository of data.
The fact that, for a fixed schema **S**, a set of values $\mathcal{V}$ a domain $\Sigma = (\mathcal{O}, \delta)$ and an interpretation for the primitive classes, the set of instances of a defined class is unambiguously determined, allows a LOGIDATA* database to *recognize* new instances of a class. So, for example, given the schema specification reported in Appendi A, if #John is defined in such a way that:

#John $\in \mathcal{I}(Person)$
$\delta(\#John) =$ [name:John birthdate:[day:25 month:7 year:1953]
emp-code:I23435 salary:2000 works-at:#Ote
prj:#LogDB budget:[prj-amount:1500]]

then the system recognizes the object #John as an Employee and a PrjLeader without explicitly asserting these facts and includes #John in the answer to a query about the instances of Employee and PrjLeader.

In addition our system deals with *incompletely specified objects* in a way that is semantically coherent. For example, we can assert that:

#Mary $\in \mathcal{I}(Student)$, with attributes:
regist-num = 876362
enrolled = #Yale
enrolled-course = {mathematics,physic}

without specifying the fillers for the name and birthdate attributes of the #Mary object (see in the Appendix A the description of the Student class) and without any explicit assertion about the #Yale object. Then, given that the fillers of the enrolled attribute in the Student class must belong to the College class, LOGIDATA* includes the #Yale object in the College class and assignes an undefined value to the unspecified properties of both #Mary and #Yale objects:

$\#\text{Yale} \quad\quad \in \mathcal{I}\,(College)$

$\delta(\#Yale) \;\; = [\text{name:nil address:nil courses:nil}]$

$\delta(\#Mary) = [\text{name:nil birthdate:nil regist-num:876362 enrolled:\#Yale}$
$\quad\quad \text{enrolled-course:}\{\text{mathematics,physic}\}]$

Therefore, LOGIDATA* deals with incrementally evolving descriptions, allowing the user to maintain a partial view of the domain of discourse; incomplete information may be gradually eliminated as new knowledge is acquired.

5 Schema Coherence and Subsumption

An interesting property of classes is disjointness [6], i.e., whether two class descriptions denote necessarily mutually exclusive sets. In LOGIDATA* two class descriptions can be *disjoint* because of non-overlapping restrictions over the same label. In the following schema:

class Person $\dot{\leq}$ [name:string birthdate:string telephone:string]
class Childless-Person $\doteq$ ISA Person [children:*EmptySet*]
class Fertile-Parent $\doteq$ ISA Person [children:$\{\text{Person}\}_{1,5}$]
class Successfull-Parent $\doteq$ ISA Person [children:$\{\text{Person}\}_{6,20}$]

Childless-Parent, Fertile-Parent and Successful-Parent are disjoint classes given that, for the label children:

$$\mathcal{I}\,(\{Person\}_{1,5}) \cap \mathcal{I}\,(\{Person\}_{6,20}) = \mathcal{I}\,(\mathit{EmptySet}) \cap \mathcal{I}\,(\{Person\}_{1,5}) =$$
$$\mathcal{I}\,(\mathit{EmptySet}) \cap \mathcal{I}\,(\{Person\}_{6,20}) = \emptyset$$

In general, the following definitionition holds.

Definition 3 (Disjoint types). *Given a schema* **S***, two types* t_1 *and* t_2 *are* disjoint *if and only if for each domain* Σ *and for each interpretation function* $\mathcal{I}$ *definitioned over* $\mathcal{V}$*,* $\mathcal{I}\,(t_1) \cap \mathcal{I}\,(t_2) \;=\; \emptyset$ *.*

An incoherent type description contains two or more disjoint types. For example, the following class descriptions are incoherent:
classC1 $\doteq$ ISA Fertile-Parent Successful-Parent
classC2 $\doteq$ ISA Person [telephone:int]

Definition 4 (Incoherent type). *Given a schema* **S** *a type* t *is* incoherent *if and only if for each domain* Σ *and for each interpretation function* $\mathcal{I}$ *definitioned over* $\mathcal{V}$*,* $\mathcal{I}\,(t) = \emptyset$*.*

We say that a schema **S** is *coherent* if it does not contain incoherent types. LOGIDATA* does not permit to create such types in order to always enforce a consistent knowledge state (see sect. 6, where the above definitionition is translated into syntactic constraints).

The introduction of the interpretation function, together with the notion of defined class, allows us to formalize the concept of *subsumption* between classes and define an algorithm for its computation.

Definition 5 (Subsumption). *Given classes C1 and C2, C2* subsumes *C1* $(C2 \sqsupseteq C1)$ *or* SUBS$(C2, C1)$*) if and only if for each domain* Σ *and for each interpretation function* $\mathcal{I}$ *defined over* $\mathcal{V}$, $\mathcal{I}(C1) \subseteq \mathcal{I}(C2)$ *holds.*

In our framework, every ISA clause corresponds to a subsumption relationship: if C1ISAC2 then C2 subsumes C1. The opposite is not necessarily true; a class can subsume another one even if subsumption is not esplicitly defined by means of an ISA clause.

Because our interpretation function is totally based on structural characteristics, the meaning of a structured description is only determined by its internal structure. This allows us to make an algorithm to deduce all the subsumption relationships among classes implicitly given by the structural conditions appearing in the class descriptions. For this purpose we introduce an ordering on $\mathcal{T}(\mathbf{L}, \mathbf{B}, \mathbf{C}, \mathbf{T})$ called *refinement* (in symbols $\leq$), based on type's syntactic features.

Definition 6 (Refinement). *Type t is a refinement of type t' ($t \leq t'$ or REF(t, t')), if and only if:*

R1 $t \in \mathbf{T} \cup \mathbf{C} \cup \mathbf{B}$ *and* $t = t'$
R2a $t \in \mathbf{T}$ *and* $\mathrm{TYP}(t) \leq t'$
R2b $t' \in \mathbf{T}$ *and* $t \leq \mathrm{TYP}(t')$
R2c $t, t' \in \mathbf{T}$ *and* $\mathrm{TYP}(t) \leq \mathrm{TYP}(t')$
R3a $t \in \mathbf{C}_P \cup \mathbf{C}_D$ *and* $t' = \top$
R3b $t \in \mathbf{C}_P \cup \mathbf{C}_D$ *and* $t' \in \mathbf{C}_P$ *and* $t' \in \Phi(t)$
R3c $t \in \mathbf{C}_P \cup \mathbf{C}_D$ *and* $t' \in \mathbf{C}_D$ *and:*
1. $\forall C'_p \in \Phi(t')$ *then* $C'_p \in \Phi(t)$*,too*
2. $\mathrm{TYP}(t) \leq \mathrm{TYP}(t')$

R4 $t = [l_1 : t_1 \ldots l_k : t_k \ldots l_{k+p} : t_{k+p}]$, $t' = [l_1 : t'_1 \ldots l_k : t'_k]$*; with* $k \geq 0$*,* $p \geq 0$*, and* $t_i \leq t'_i$ *, for* $i=1, \ldots, k$
R5 $t = \{t_1\}_{m,n}$ *and* $t' = \{t'_1\}_{p,q}$*, with* $t_1 \leq t'_1$*,* $m \geq p$ *and* $n \leq q$
R6 $t = EmptySet$ *and* $t' = \{t'_1\}_{0,m}$*;* $\forall t'_1 \in \mathcal{T}$ *and* $\forall m > 0$*.*

The following theorem, based on refinement, characterizes subsumption syntactically.

Theorem 1. *Given classes C1 and C2, C2* subsumes *C1 (*SUBS*(C2,C1) or* $C1 \sqsubseteq C2$*) if and only if C1 refines C2 (i.e.,* $C1 \leq C2$*).*

Proof. The proof that the theorem agrees with the above given subsumption definitionition (i.e., sound and complete) is in Appendix B.

Due to the formalism previously introduced (i.e., syntax + semantics + subsumption), we obtain an object-oriented data model based on *classification*, allowing the *self organization* of the classes defined into the subsumption hierarchy. In other words, all the subsumption relationships are computed by the system, independently of user-given ISA links. With reference to the schema shown in the Appendix A, a classification algorithm finds out the following subsumption relationships:

$$RschProject \sqsubseteq Project, PrjLeader \sqsubseteq Employee, Secretary \sqsubseteq Employee$$
$$PrjManager \sqsubseteq Employee, Attending - Student \sqsubseteq Student$$

6 Syntactic Constraints for Coherence

In this section we characterize the notion of coherent type from a syntactic point of view by means of giving the definition of *correct type description*. Furthermore, as regards multiple inheritance, some syntactic constraints are added to a class description using the ISA clause in order for the type of the subclass elements to be compatible with the superclass type [3].

The following definition gives rise to an algorithm looking for correct type descriptions that can be shown to be consistent with respect to the language semantics (see sect. 3).

Definition 7 (Correct type description).

1. $t = EmptySet$ *or* $t = \top$
   the description of t *is correct.*
2. $t = \mathrm{B}_i$, *with* $\mathrm{B}_i \in \mathbf{B}$
   the description of t *is correct.*
3. $t = [\mathrm{l}_1 : t_1, \ldots, \mathrm{l}_n : t_n]$
   the description of t *is correct if* $l_i \neq l_j$ $\forall$*i,j=1, . . . ,n with i≠j, and each* t_i *is a correct type description.*
4. $t = \{t'\}_{min,max}$
   the description of t *is correct if* t' *is a correct type description, min≤max and max>0.*
5. **type** $t = t'$ *(where* $t =$<*type-id*>*)*
   the description of t *is correct if* t' *is a correct set or tuple type description.*
6. **class** C $< prim - def >$ ISA $C_1 \ldots C_n$ $< tuple - type >$
   the description of class C *is correct if* <*tuple-type*> *is a correct type description. Furthermore, for labels appearing in some* C_i *belonging to the* ISA *clause and in* <*tuple-type*>*, the type associated with such labels in* <*tuple-type*> *must be a refinement of the type associated with the omologous labels in* C_i*. Moreover, if the same label appears in more than one* C_i *with different types, then it must explicitly appear in* <*tuple-type*> *with a type that is the refinement of all the previous ones.*

The following theorem shows the soundness of the above given definition with respect to the definition of incoherent type (see sect. 5).

Theorem 2 (Soundness of correct type description). *Given a schema* **S***, let* $\mathcal{T}$ *the set of type descriptors and* t *a correct type description in* $\mathcal{T}$*, then there exists a domain* Σ *and an interpretation function* $\mathcal{I}$ *defined over* $\mathcal{V}$ *such that* $\mathcal{I}(t) \neq \emptyset$.

For example, the definition of RschLeader is correct because the *prj* label appears in both PrjLeader and the tuple type associated with RschLeader, and $\{\text{RschProject}\}_{1,2} \leq \{\text{Project}\}_{1,3}$. The definition of Supervisor shows an example of multiple inheritance; the *budget* label is present in Supervisor with a type that refines both types appearing in PrjLeader and PrjManager, respectively.

We can now characterize syntactically the function TYP, with respect to both the definition of semantics (see sect. 3) and the above given definition of correct type description. Given a coherent schema **S**, TYP associates with each symbol $S \in \mathbf{C} \cup \mathbf{T}$ the type descriptor defined by the expansion of the non-terminal symbol appearing in the corresponding declaration, with the following exception: if a class is described by means of an ISA clause, then TYP(C) is a tuple type with all the labels that appear in the superclass descriptions and in the tuple type that describes the class. In the case that the same label appears in the tuple type that describes the class and in one or more superclass descriptions, then TYP(C) presents such label as it is defined in the tuple type describing the class.

Using ISA in a class description makes it possible to exploit inheritance, i.e., the mechanism that allows a class to inherit properties of its superclasses. From the above definitions, it is clear that *multiple inheritance* is only allowed in a framework of "re-definition", and re-definition is subjected to some type refinement constraints; a similar approach also appears in O_2 [8]. We show some results obtained by the function TYP evaluation:

TYP(Date) = [day:int month:int year:int]
TYP(RschLeader) = [name:string birthdate:Date emp-code:string
salary:int works-at:Company prj:$\{\text{RschProject}\}_{1,2}$
budget:[prj-amount:int]]
TYP(Supervisor) = [name:string birthdate:Date emp-code:string
salary:int works-at:Company prj:$\{\text{Project}\}_{1,3}$
has-secretary:Secretary budget:[prj-amount:int
external-amount:int budget-plan:int]]
TYP(Person) = [name:string birthdate:Date]
TYP(Woman) = [name:string birthdate:Date].

7 Computational complexity

In the environment of KL-ONE-like languages, particular attention is given to studying the relationships existing between the model definition language's expressive power and the related subsumption problem's complexity [7][9][12] in order to obtain tractable systems.

As far as the complexity of SUBS(C1,C2) is concerned, we can repeat the same considerations made in [10] about the intractability of terminological reasoning. As a matter of facts, in the worst case, SUBS(C1,C2) performs subsequent expansions of both classes until it is possible to calculate SUBS($\widehat{C}_1$,$\widehat{C}_2$), with $\widehat{C}_1$ and $\widehat{C}_2$ being *completely expanded classes*, where $\widehat{C}$ is a completely expanded class if TYP($\widehat{C}$) does not contain any type-names or class-names: these names are replaced by their descriptions. The transformation from C into $\widehat{C}$ is possible in a finite number of steps because our language does not have any recursive definitions (acyclic terminology), but the trouble is that it induces expressions of size $O(m^n)$, where m is the size of the unexpanded class C [10][1], as the following example shows. Let B_0 be a base type and:

class C_1 [l:B_0 l':B_0]
class C_2 [l:C_1 l':C_1]
... ...
class C_n [l:C_{n-1} l':C_{n-1}]

In this case the size of $\widehat{C}_n$ is $O(2^n)$.

Now we define two measures accounting for the names expansion process inside every type and we show that the subsumption algorithm is tractable with respect to these measures.

Definition 8 (Depth of a Type). *We indicate by d(t) the depth of a generic type:*

- *If* $t = \mathrm{B}$ *is a basic type, d(t)=0*
- *If* $t = EmptySet$*, d(t)=0*
- *If* $t = \top$*,* $d(\top)=0$
- *If* $t = [l_1 : t_1, \ldots, l_m : t_m]$*, d(t)=1+max{d(ti), with i=1,...,m}*
- *If* $t = \{t_1\}_{min,max}$*, d(t)=1+d(t_1)*
- *If* $t = C$ *is a class name, d(t)=d(*TYP(C)*)*
- *If* $t = T$ *is a type name, d(t)=d(*TYP(T)*).*

Definition 9 (Expanded Size of a Type). *Let t be a generic type, we indicate by* $\mid t \mid$ *the* expanded size *of t:*

- *If* $t = \mathrm{B}$ *is a basic type,* $\mid t \mid$*=1*
- *If* $t = EmptySet$*,* $\mid EmptySet \mid$*=1*
- *If* $t = \top$*, then* $\mid \top \mid$*=1*
- *If* $t = [l_1 : t_1, \ldots, l_m : t_m]$*,* $\mid t \mid = \sum_{i=1}^{m}(1+ \mid t_i \mid)$
- *If* $t = \{t_1\}_{min,max}$*,* $\mid t \mid = 1+ \mid t_1 \mid$*;*
- *If* $t = C$ *is a class name,* $\mid C \mid = \parallel \Phi(C) \parallel + \mid \mathrm{TYP}(C) \mid$
- *If* $t = T$ *is a type name,* $\mid T \mid = \mid \mathrm{TYP}(T) \mid$*.*

The following theorem can be proven by induction on type depth (we denote the algorithm that calculates if $t \leq t'$ by means of REF(t,t')).

Theorem 3 (Subsumption Algorithm is Polynomial). *Given two classes* C_1 *and* C_2 SUBS(C_1, C_2) *runs in* $O(\mid C_1 \mid \times \mid C_2 \mid)$ *time, i.e. is* polynomial *in the expanded size of a class.*

Proof. Since SUBS(C_1, C_2)=TRUE if and only if REF(C_1, C_2)=TRUE, we prove that REF(C_1, C_2) runs in $O(\mid C_1 \mid \times \mid C_2 \mid)$ time; further, since that the worst case happens only in the presence of defined classes, we suppose that in every computation of REF(C_i, C_j) C_j is a defined class. REF(C_1, C_2) is true if and only if:

– $\Phi(C_2) \subseteq \Phi(C_1)$; this is done in $O(\parallel \Phi(C_1) \parallel \times \parallel \Phi(C_2) \parallel)$ steps
– $\mathrm{REF}(\mathrm{TYP}(C_1), \mathrm{TYP}(C_2))$, with
$\mathrm{TYP}(C_1) = [l_1 : t_1, \ldots, l_n : t_n]$
$\mathrm{TYP}(C_2) = [l_1 : t_1, \ldots, l_m : t_m]$

Given that $\mathrm{d}(C_2)$=$\mathrm{d}(\mathrm{TYP}(C_2))$, the proof proceeds by induction on the depth of $\mathrm{TYP}(C_2)$.

1. If $\mathrm{d}(\mathrm{TYP}(C_2))$=1 then $\mathrm{d}(t'_i)$=0 $\forall t'_i$ in $\mathrm{TYP}(C_2)$ ($\mathrm{d}(C_2)$=1 and $\mid C_2 \mid = m+ \parallel \Phi(C_2) \parallel$) , then for each label l'_i the $\mathrm{REF}(\mathrm{TYP}(C_1), \mathrm{TYP}(C_2))$ procedure must scan all the l_j appearing in $\mathrm{TYP}(C_1)$ (at most $\mathrm{TYP}(C_1)$ steps), looking for a label $l_j = l'_i$ with $t_j = t'_i$ or $t_j \in \mathbf{C}$ if $t'_i = \top$; this is done in $O(\mid \mathrm{TYP}(C_1) \mid \times \mid \mathrm{TYP}(C_2) \mid)$ steps, then $\mathrm{REF}(C_1, C_2)$ runs in $O(\mid C_1 \mid \times \mid C_2 \mid)$ time.
2. If $\mathrm{d}(\mathrm{TYP}(C_2))$=2 ($\max\{\mathrm{d}(t_i)\}$=1); then, for each label l'_i in $\mathrm{TYP}(C_2)$ we must find the corresponding l_j in $\mathrm{TYP}(C_1)$ (at most $\mid \mathrm{TYP}(C_1) \mid$ steps) and then call $\mathrm{REF}(t_j, t'_i)$ with $\mathrm{d}(t'_i) \leq 1$; the following cases hold:
   (a) $\mathrm{d}(t'_i)$=0, then $\mathrm{REF}(t_j, t'_i)$ takes $O(\mid t_j \mid \times \mid t'_i \mid)$ steps.
   (b) $\mathrm{d}(t'_i)$=1 with t'_i tuple type. Let:
   $t_j = [l_{j1} : t_{j1} \ldots l_{jn} : t_{jn}]$ and $t'_i = [l'_{i1} : t'_{i1} \ldots l'_{im} : t'_{im}]$
   In this case, given that $\mathrm{d}(t'_{ik})$=0 $\forall k = 1, \ldots, m$, the analysis is similar to that of point 1; then $\mathrm{REF}(t_j, t'_i)$ takes $O(\mid t_j \mid \times \mid t'_i \mid)$ steps.
   (c) $\mathrm{d}(t'_i)$=1 with t'_i set type. Let:
   $t_j = \{t_{j1}\}_{m,n}$ or $t_j = \mathit{EmptySet}$ and $t'_i = \{t'_{i1}\}_{p,q}$ with $\mathrm{d}(t'_{i1})$=0.
   There must be: a) m$\geq$p, n$\leq$q and $\mathrm{REF}(t_{j1}, t'_{i1})$; or b) p=0. In both cases $\mathrm{REF}(t_j, t'_i)$ takes $O(\mid t_j \mid \times \mid t'_i \mid)$ steps.
   (d) $\mathrm{d}(t'_i)$=1 with t'_i class name. There must be:
   a) $\Phi(t'_i) \subseteq \Phi(t_j)$; this is done in $O(\parallel \Phi(t'_i) \parallel \times \parallel \Phi(t_j) \parallel)$ steps
   b) $\mathrm{REF}(\mathrm{TYP}(t_j), \mathrm{TYP}(t'_i))$, where $\mathrm{TYP}(t'_i)$ is a tuple type with $\mathrm{d}(\mathrm{TYP}(t'_i))$ = 1. Then as showed in point 1., $\mathrm{REF}(\mathrm{TYP}(t_j),\ \mathrm{TYP}(t'_i))$ takes $O(\mid \mathrm{TYP}(t_j) \mid \times \mid \mathrm{TYP}(t'_i) \mid)$ steps. Given that, for a class name $\mid C \mid = \parallel \Phi(C) \parallel + \mid \mathrm{TYP}(C) \mid$, then the total effort for computing $\mathrm{REF}(t_j, t'_i)$ is $O(\mid t_j \mid \times \mid t'_i \mid)$.

   In all cases examinated we have that $\mathrm{REF}(t_j, t'_i)$ runs in $O(\mid t_j \mid \times \mid t'_i \mid)$ steps, then the total effort for the m labels is:

$$\sum_{i=1}^{m} (\mid \mathrm{TYP}(C_1) \mid + \mid t_j \mid \times \mid t'_i \mid) \ \leq \mid \mathrm{TYP}(C_1) \mid \times \sum_{i=1}^{m} (1 + \mid t'_i \mid) \ =$$

$$\mid \mathrm{TYP}(C_1) \mid \times \mid \mathrm{TYP}(C_2) \mid$$

   and $\mathrm{REF}(C_1, C_2)$ runs in $O(\mid C_1 \mid \times \mid C_2 \mid)$ steps.
3. Let us assume that for $\mathrm{d}(\mathrm{TYP}(C_2))$=k ($\mathrm{d}(C_2)$=k), $\mathrm{REF}(\mathrm{TYP}(C_1), \mathrm{TYP}(C_2))$ runs in $O(\mid \mathrm{TYP}(C_1) \mid \times \mid \mathrm{TYP}(C_2) \mid)$ steps. This implies that:
   (a) $\mathrm{REF}(C_1, C_2)$ with $\mathrm{d}(C_2)$=k runs in $O(\mid C_1 \mid \times \mid C_2 \mid)$ time
   (b) Given that $\mathrm{TYP}(C_2)$ is a generic tuple type, then $\mathrm{REF}(t_1, t_2)$ with t_1, t_2 tuple types and $\mathrm{d}(t_2)$=k runs in $O(\mid t_1 \mid \times \mid t_2 \mid)$ steps

(c) If t'_i is a set type with d(t'_i)=k-1, then REF(t_j, t'_i) runs in $O(\mid t_j \mid \times \mid t'_i \mid)$ steps.

4. Suppose that d(TYP(C_2))=k+1, then max{d(t'_i) | t'_i is in TYP(C_2)}=k (d(C_2)=k+1); then, for each label l'_i in TYP(C_2) we must find the corresponding l_j in TYP($C1$) (at most TYP(C_1) steps) and then recursively call REF(t_j, t'_i) with d(t'_i) $\leq$k; the following cases hold:
   (a) t'_i is a class name or a tuple type; then, by induction, REF(t_j, t'_i) with d(t'_i)$\leq$k runs in $O(\mid t_j \mid \times \mid t'_i \mid)$ steps
   (b) t'_i is a set type with d(t'_i)=k. Let:
   $t_j = \{t_{j1}\}_{m,n}$ or $t_j = EmptySet$ and $t'_i = \{t'_{i1}\}_{p,q}$ with d(t'_{i1})=k-1.
   There must be: a) m$\geq$p ,n$\leq$q and REF(t_{j1}, t'_{i1}) with d(t'_{i1})=k-1; or b) p=0. In both cases, by induction, REF(t_j, t'_i) takes $O(\mid t_j \mid \times \mid t'_i \mid)$ steps.

   As shown in the conclusion of point 2., REF(TYP(C_1), TYP(C_2)) runs in $O(\mid$ TYP$(C_1) \mid \times \mid$ TYP$(C_2) \mid)$ steps, then REF(C_1, C_2) runs in $O(\mid C_1 \mid \times \mid C_2 \mid)$ steps.

8 Conclusions

Our study aims at endowing an object-oriented database model with the inference capabilities related to subsumption. Subsumption and its applications have been throughly investigated in the field of Knowledge Representation systems. While database models usually refer to different deductive paradigms, such as logic, recent studies pointed out usefulness of taxonomic reasoning for dealing with data management at both an intensional and extensional level. The main result of our paper is to show how this capability can be added to an object-oriented data model, proposed in a database environment, increasing then its features still maintaining the original ones; as a matter of fact, we develop a formal framework for treating values and objects, and types and classes in a well-founded and uniform way that agrees with taxonomic reasoning. Our future work will deal with managing extensional knowledge, investigating the features provided at this level by taxonomic reasoning, in particular we will investigate the instance recognition problem. To date, the Knowledge Representation Systems emphasized the intensional level aspect of the Knowledge Base, while is our opinion that providing the system with inferential tecniques acting both at an intensional and extensional level will improve the system efficiency and usability.

References

1. S. Abiteboul and R. Hull. IFO: A Formal Semantic Database Model. In *ACM TODS*, vol. 12, n.4, 1987.
2. D. Beneventano, S. Bergamaschi, C. Sartori and A. Artale, F. Cesarini, G. Soda. Taxonomic Reasonig in LOGIDATA$^+$. To appear in the LOGIDATA book, Springer Verlag.
3. P. Atzeni, L. Tanca. The LOGIDATA$^+$ model and Language. In *Workshop on Information Systems '90*, Kiev, Oct. 1990, to appear in Lecture Notes in Computer Science, Springer-Verlag.

4. R.J. Brachman, D.L. MacGuinness, P.F. Patel-Schneider and L.A. Resnick. Living with CLASSIC: When and How to Use a KL-ONE-Like Language. In *Principles of Semantic Networks*, J.Sowa, Morgan Kaufmann Publishers, Inc., 1990.
5. L. Cardelli. A Semantics of Multiple Inheritance. *Semantics of Data Types*, Springer-Verlag, pp.51-67, 1984.
6. L.M.L. Delcambre and K.C. Davis. Automatic Validation of Object-Oriented Database Structures. In *Proceedings of Int. Conf. on Data Engineering*, 1989.
7. F.M. Donini, M. Lenzerini, D. Nardi, W. Nutt. The complexity of concept languages. In Proceedings of the 2nd Int. Conf. on Principles of Knowledge Representation and Reasoning, KR-91, Morgan Kaufmann, 1991.
8. C. Lecluse and P. Richard. Modelling Complex Structures in Object-Oriented Databases. In *ACM PODS*, pp.360-367,1 1989.
9. H.J. Levesque and R.J. Brachman. Expressiveness and Tractability in Knowledge Representation and Reasoning. In *Computational Intelligence*, 3:78-93, 1987.
10. B. Nebel. Terminological Reasoning is Inherently Intractable. In *IWBS Report 82*, September 1989.
11. B. Nebel. Reasoning and Revision in Hybrid Representation Systems. In *Lecture Notes in Artificial Intelligence*, n. 422, Springer-Verlag, 1990.
12. P.F. Patel-Schneider. Undecidability of subsumption in NIKL. In *Artificial Intelligence*, 39:263-272, 1989.

A Appendix

In this Appendix we present an example of schema specification, using the LOGIDATA* syntax, along with an example of schema interpretation. The class and type names are indicated with a capital letter.

<u>Schema specification</u>

type Date	$=$	[day:int month:int year:int]
class Organization	$\dot{\leq}$	[name:string residence:string]
class Consortium	$\dot{\leq}$	[members:$\{$Organization$\}_{1,\infty}$]
class Project	$\doteq$	[proj-code:string description:string]
class RschProject	$\doteq$	[proj-code:string description:string has-units:Consortium]
class College	$\doteq$	[name:string address:$\{$string$\}_{1,\infty}$ courses:$\{$string$\}_{1,\infty}$]
class University	$\doteq$	ISA Organization [has-colleges:$\{$College$\}_{1,\infty}$]
class Company	$\doteq$	ISA Organization [emp-num:int turnover:int]
class Person	$\dot{\leq}$	[name:string birthdate:Date]
class Woman	$\dot{\leq}$	ISA Person
class Student	$\doteq$	ISA Person [regist-num:int enrolled:College enrolled-course:$\{$string$\}$]
class Attending-Student	$\doteq$	ISA Person [regist-num:int enrolled:College enrolled-course:$\{$string$\}_{2,10}$]
class Employee	$\doteq$	ISA Person [emp-code:string salary:int works-at:Company]
class Unemployed	$\doteq$	ISA Person [works-at:EmptySet]
class Secretary	$\dot{\leq}$	ISA Woman [emp-code:string salary:int works-at:Company]
class PrjLeader	$\doteq$	ISA Person [emp-code:string salary:int works-at:Company prj:$\{$Project$\}_{1,3}$ budget:[prj-amount:int]]
class RschLeader	$\doteq$	ISA PrjLeader [prj:$\{$RschProject$\}_{1,2}$]
class PrjManager	$\doteq$	ISA Person [emp-code:string salary:int works-at:Company has-secretary:Secretary budget:[prj-amount:int external-amount:int]]
class Supervisor	$\doteq$	ISA PrjLeader PrjManager [budget:[prj-amount:int external-amount:int budget-plan:int]]

<u>Schema interpretation</u>

Given the following domain:
$\mathcal{O}$ = {#Eng,#Med,#Art,#FI-univ,#Ote,#c1,#LogDB,#OODB,#Anne,#Al,

#Roland,#Robert,#Max}

$\delta(\#Eng)$	= [name:Engineering address:{Via,S.Marta,3,FI} courses:nil]
$\delta(\#Med)$	= [name:Medicine address:{Viale,Morgagni,1,FI} courses:nil]
$\delta(\#Art)$	= [name:Liberal-Arts adderss:{Piazza, Brunelleschi,FI} courses:nil]
$\delta(\#FI-univ)$	= [name:Universita'-Firenze residence:Firenze has-colleges:{#Eng,#Med,#Art}]
$\delta(\#Ote)$	= [name:Ote-Biomedica residence:Firenze emp-num:1500 turnover:1000]
$\delta(\#c1)$	= [members:{#FI-univ,#Ote}]
$\delta(\#LogDB)$	= [prj-code:DB14 description:deductive-database]
$\delta(\#OODB)$	= [prj-code:DB15 description:O-O-database has-units:#c1]
$\delta(\#Anne)$	= [name:Anne birthdate:[day:22 month:11 year:1964] emp-code:I76009 salary:1200 works-at:#Ote]
$\delta(\#Al)$	= [name:Alexander birthdate:[day:25 month:7 year:1964] emp-code:I76010 salary:2000 works-at:#Ote prj:{#LogDB} budget:[prj-amount:1500]]
$\delta(\#Roland)$	= [name:Roland birthdate:[day:2 month:6 year:1947] emp-code:I302 salary:4000 works-at:#Ote has-secretary:#Anne prj:{#LogDB} budget:[prj-amount:1500 external-amount:500 budget-plan:3000]]
$\delta(\#Robert)$	= [name:Robert birthdate:[day:7 month:5 year:1954] emp-code:I205 salary:2000 works-at:#Ote prj:{#OODB} budget:[prj-amount:1000]]
$\delta(\#Max)$	= [name:Max birthdate:[day:nil month:7 year:1969] regist-num:176487 enrolled:#Eng enrolled-course:{mathematics,physic, chemistry}]

Fixed the interpretation for the primitive classes:

$\mathcal{I}(Organization)$	= {#FI-univ,#Ote}
$\mathcal{I}(Consortium)$	= {#c1}
$\mathcal{I}(Person)$	= {#Anne,#Al,#Roland,#Robert,#Max}
$\mathcal{I}(Woman)$	= {#Anne}
$\mathcal{I}(Secretary)$	= {#Anne}

The interpretation of the definite classes is unambiguously determined:

$\mathcal{I}(University) = \{\#\text{FI-univ}\}$
$\mathcal{I}(Company) = \{\#\text{Ote}\}$
$\mathcal{I}(College) = \{\#\text{Eng},\#\text{Med},\#\text{Art}\}$
$\mathcal{I}(Project) = \{\#\text{LogDB},\#\text{OODB}\}$
$\mathcal{I}(RschProject) = \{\#\text{OODB}\}$
$\mathcal{I}(Employee) = \{\#\text{Anne},\#\text{Al},\#\text{Roland},\#\text{Robert}\}$
$\mathcal{I}(Unemployed) = \{\}$
$\mathcal{I}(PrjLeader) = \{\#\text{Al},\#\text{Roland}\}$
$\mathcal{I}(PrjManager) = \{\#\text{Roland}\}$
$\mathcal{I}(Supervisor) = \{\#\text{Roland}\}$
$\mathcal{I}(RschLeader) = \{\#\text{Robert}\}$
$\mathcal{I}(Student) = \{\#\text{Max}\}$
$\mathcal{I}(Attending - Student) = \{\#\text{Max}\}$.

B Appendix

We prove that the SUBS algorithm calculates subsumption between classes. First, we must prove that if SUBS(C_1, C_2)=TRUE then $\mathcal{I}(C_2) \subseteq \mathcal{I}(C_1)$ (*soundness*); then, we must prove that if $\mathcal{I}(C_2) \subseteq \mathcal{I}(C_1)$ then SUBS(C_1, C_2)=TRUE (*completeness*). Note that the system does not have cyclic references; therefore, it is possible to use the induction principle in the proofs.

Theorem 4 (Soundness of the SUBS Algorithm). *Let C_1 and C_2 be two classes; if SUBS(C_1, C_2)=TRUE, then for each domain Σ, for each set of values $\mathcal{V}$ and for each interpretation function $\mathcal{I}$ over $\mathcal{V}$: $\mathcal{I}(C_2) \subseteq \mathcal{I}(C_1)$.*

Proof. Since SUBS(C_1, C_2)=TRUE if and only if REF(C_2, C_1)=TRUE, we prove that, given two types t and t', if $t \leq t'$ then $\forall \Sigma, \mathcal{V}$ and $\forall \mathcal{I}$ over $\mathcal{V}$: $\mathcal{I}(t) \subseteq \mathcal{I}(t')$. Assuming that $t \leq t'$, one of the following cases holds (see sect. 5):

R1,R3a-b Trivial.

R4 $t = [l_1 : t_1, \ldots, l_m : t_m]$ and $t' = [l_1 : t'_1, \ldots, l_n : t'_n]$.
with t defined at least on all the labels appearing in t' and such that $\forall l'_i$, i=1, $\ldots$,n, $\exists l_j$ for which $l'_i = l_j$ and REF(t_j, t'_i)=TRUE. Therefore, by induction, $\mathcal{I}(t_j) \subseteq \mathcal{I}(t'_i)$ and, because of point 2 of the interpretation function: $\mathcal{I}(t) \subseteq \mathcal{I}(t')$.

R5 $t = \{t_1\}_{m,n}$ and $t' = \{t'_1\}_{p,q}$, with REF(t_1, t'_1)=TRUE, m$\geq$p and n$\leq$q.
By induction, $\mathcal{I}(t_1) \subseteq \mathcal{I}(t'_1)$; therefore, due to point 3 of the interpretation function: $\mathcal{I}(t) \subseteq \mathcal{I}(t')$.

R2a-b-c t and/or $t' \in \mathbf{T}$.
Assuming t_1 = TYP(t) and/or t'_1 = TYP(t'), we have $t_1 \leq t'_1$, whith t_1 and t'_1 tuple type or set type. Due to the previous points R4. and R5., $\mathcal{I}(t_1) \subseteq \mathcal{I}(t'_1)$; then, for point 4 of the interpretation definition: $\mathcal{I}(t) \equiv \mathcal{I}(t_1)$ and $\mathcal{I}(t') \equiv \mathcal{I}(t'_1)$; therefore $\mathcal{I}(t) \subseteq \mathcal{I}(t')$.

R6 $t = EmptySet$ and $t' = \{t'_1\}_{0,m}$.
$\mathcal{I}(EmptySet) = \{\emptyset\}$, but $\emptyset \in \mathcal{I}(\{t'_1\}_{0,m})$ $\forall t'_1 \in \mathcal{T}$ and $\forall m > 0$; therefore $\mathcal{I}(EmptySet) \subseteq \mathcal{I}\left(\{t'_1\}_{0,m)}\right)$.

R3c t and $t' \in \mathbf{C}$ with $t' \in \mathbf{C}_D$ and:
- $\Phi(t') \subseteq \Phi(t)$
- REF(TYP(t), TYP(t'))=TRUE

Therefore, by induction $\mathcal{I}(\mathrm{TYP}(t)) \subseteq \mathcal{I}(\mathrm{TYP}(t'))$; then, due to point 5.b of the interpretation definition, $\mathcal{I}(t) \subseteq \mathcal{I}(t')$.

The *completeness* of the SUBSalgorithm can be proven on the basis of the following two lemmas. The first lemma allows us to construct an interpretation function in such a way that a particular value v_1 is in the interpretation of each tuple type except for a particular exception.

Lemma 1. *Let* $\mathbf{L}$ *be a set of labels,* $l^\circ \in \mathbf{L}$*;* t *a generic type;* $\mathcal{V}$ *a set of values. Moreover, let* $\mathcal{I}$ *be an interpretation function over* $\mathcal{V}$*;* v_1 *and* $v^\circ \in \mathcal{V}$ *such that:*

- $v^\circ \in \mathcal{I}(t)$
- v_1 *is a tuple value*
- $v_1(l_i) = nil$ $\forall l_i \in \mathbf{L}$, *for* $l_i \neq l^\circ$
- $v_1(l^\circ) = v^\circ$.

Then, for every tuple type:
$x = [l_1 : t_1, \ldots, l_k : t_k]$
if x does not *contain* $l^\circ : u$, *with* u *such that REF*(t, u)*=FALSE, then* $v_1 \in \mathcal{I}(x)$.

Proof. There are two cases.

1. The l° label is not present in x.
   Due to our construction, v_1 is surely defined over all the labels appearing in x and $v_1(l_i) \in \mathcal{I}(t_i) \cup \{nil\}$ $\forall$ i=1, ...,k. Then, for point 2 of the interpretation definition, $v_1 \in \mathcal{I}(x)$.
2. The l° label is present in x:
   $x = [l_1 : t_1 \ldots l^\circ : s \ldots l_k : t_k]$ with s generic type such that $t \leq s$.
   It is enough to prove that $v_1(l^\circ) \in \mathcal{I}(s)$. Due to our construction, $v_1(l^\circ) = v^\circ \in \mathcal{I}(t)$; but we have $\mathcal{I}(t) \subseteq \mathcal{I}(s)$ because $t \leq s$, then $v_1(l^\circ) \in \mathcal{I}(s)$. Therefore, $v_1 \in \mathcal{I}(x)$.

Lemma 2. *For each* $t \in \mathcal{T}(\mathbf{L}, \mathbf{B}, \mathbf{C}, \mathbf{T})$ *it is possible to find* n *distinct values* $v_1, \ldots, v_n \in \mathcal{V}$ *and an interpretation function* $\mathcal{I}$ *defined over* $\mathcal{V}$ *such that* $v_1, \ldots, v_n \in \mathcal{I}(t)$.

Proof. The proof follows from the consideration that, for each basic type B_i, $\mathcal{I}(\mathrm{B}_i)$ is a countable set. Therfore, it's easy to show by induction that the lemma holds.

Now we can prove the *completeness* of the subsumption algorithm.

Theorem 5 (Completeness of the SUBS Algorithm). *Given two classes C_1 and C_2, if $\mathcal{I}(C_1) \subseteq \mathcal{I}(C_2)$, for every set of values $\mathcal{V}$ and for every interpretation function $\mathcal{I}$ over $\mathcal{V}$, then SUBS(C_2, C_1)=TRUE.*

Proof. Given that SUBS(C_2, C_1) is true if and only if REF(C_1, C_2) is true, we show that, given two generic types x and y, if REF(x, y)=FALSE, it is always possible to find a set $\mathcal{V}$ and an interpretation function $\mathcal{I}$ over $\mathcal{V}$ such that: $\mathcal{I}(x) \not\subset \mathcal{I}(y)$.

1. If x, y have different types (for example, x is a tuple type and y a set type), then $\mathcal{I}(x) \cap \mathcal{I}(y) = \emptyset$ because the sets $\mathcal{D}, \mathcal{O}, \mathcal{V}_{\mathrm{T}}, \mathcal{V}_{\mathrm{S}}$ are pairwise disjoint; therefore, $\mathcal{I}(x) \not\subset \mathcal{I}(y)$.
2. If $x, y \in \mathbf{B}$ (they are base types), REF(x, y)=FALSE $\Leftrightarrow x \neq y$. Then $\mathcal{I}(x) \not\subset \mathcal{I}(y)$, because the base types are pairwise disjoint.
3. If x, y are tuple types, we have two different cases.
   (a) y contains $l^\circ : u$, while x does not contain the l° label.
   We build a tuple value v_1 defined exclusively on $\mathbf{L} - \{l^\circ\}$ and such that $v_1(l_i) = nil\ \forall l_i \in \mathbf{L} - \{l^\circ\}$. Let $v_1 \in \mathcal{V}$ and $\mathcal{I}$ an interpretation function over $\mathcal{V}$, then $v_1 \in \mathcal{I}(x)$ but $v_1 \notin \mathcal{I}(y)$.
   (b) y contains $l^\circ : u$, x contains $l^\circ : t$ but t is not a refinement of u.
   Since REF(t, u)=FALSE, by induction, there are $\mathcal{I}^*, \mathcal{V}^*$ and $v^* \in \mathcal{V}^*$ such that $v^* \in \mathcal{I}^*(t)$ but $v^* \notin \mathcal{I}^*(u)$. Now we build a tuple value v_1 such that $v_1(l_i) = nil\ \forall l_i \in \mathbf{L} - \{l^\circ\}$ and $v_1(l^\circ) = v^*$. Let $\mathcal{V} = \mathcal{V}^* \cup \{v_1\}$ (with $v_1 \neq v^*$); $\mathcal{I}$ an interpretation function over $\mathcal{V}$ such that, for every type s, $\mathcal{I}(s) \subseteq \mathcal{I}^*(s) \cup \{v_1\}$, and in particular $v^* \in \mathcal{I}(t)$. Then $v_1(l^\circ) = v^* \notin \mathcal{I}(u)$; therefore, $v_1 \notin \mathcal{I}(y)$. On the other hand, assuming $v^\circ = v^*$, the conditions of Lemma 14 (case 2) are satisfied, indeed $v_1 \in \mathcal{I}(x)$. Then $\mathcal{I}(x) \not\subset \mathcal{I}(y)$.
4. If x, y are set types with $x = \{t\}_{n,m}$ and $y = \{u\}_{p,q}$ and m,p>0, we have two different cases:
   (a) REF(t, u)=FALSE.
   By induction, there are $\mathcal{I}^*, \mathcal{V}^*,\ and\ v^* \in \mathcal{V}^*$ such that $v^* \in \mathcal{I}^*(t)$ but $v^* \notin \mathcal{I}^*(u)$. Let v_0 be a set value such that $v_0 = \{v^*, v_1, \ldots, v_{n-1}\}$; $\mathcal{V} = \mathcal{V}^* \cup \{v_0, v_1, \ldots, v_{n-1}\}$; $\mathcal{I}$ an interpretation function over $\mathcal{V}$ such that $v_i \in \mathcal{I}(t)$ $\forall$i=1, ...,n-1 (follows from lemma 15) and for every type s, $\mathcal{I}(s) \subseteq \mathcal{I}^*(s) \cup \{v_0, v_1, \ldots, v_{n-1}\}$. Then $v^* \in \mathcal{I}(t)$, while $v^* \notin \mathcal{I}(u)$. Therefore, since $v_0 = \{v^*, v_1, \ldots, v_{n-1}\} \subseteq \mathcal{I}(t)$ and $\| v_0 \| = n$ then, due to the interpretation of set type, $v_0 \in \mathcal{I}(x)$ but $v_0 \notin \mathcal{I}(y)$. Then $\mathcal{I}(x) \not\subset \mathcal{I}(y)$.
   (b) n<p or m>q.
   Let n<p, $v_1, \ldots, v_n \in \mathcal{V}^*$, $\mathcal{I}^*$ such that $v_1, \ldots, v_n \in \mathcal{I}^*(t)$ (follows from lemma 15). Let $v_0 = \{v_1, \ldots, v_n\}$; $\mathcal{V} = \mathcal{V}^* \cup \{v_0\}$; $\mathcal{I}$ an interpretation function over $\mathcal{V}$ such that for every type s, $\mathcal{I}(s) \subseteq \mathcal{I}^*(s) \cup \{v_0\}$. Then, $v_0 \subseteq \mathcal{I}(t)$ and $\| v_0 \| = n$, therefore $v_0 \in \mathcal{I}(x)$ but $\| v_0 \| < p$ then $v_0 \notin \mathcal{I}(y)$. The case m>q is similar.

5. $x = EmptySet$ and $y = \{t\}_{n,m}$ with n$\geq$1.
$\mathcal{I}\,(EmptySet) = \{\emptyset\} \not\subset \{v_s \in \mathcal{V}_{\mathrm{S}} \mid v_s \subseteq \mathcal{I}\,(t)\text{and } m \leq \parallel v_s \parallel \leq n\}$ then $\mathcal{I}\,(x) \not\subset \mathcal{I}\,(y)$ ($\parallel \emptyset \parallel = 0$).
6. If x, y are two classes, we have the following cases:
   (a) $y \in \mathbf{C}_P$ and there is not an explicit ISA such that xISA y.
   Given the semantics of a primitive class, it is alway possible to build $\mathcal{I}$,$\mathcal{V}$ and $o \in \mathcal{O} \subseteq \mathcal{V}$ such that $o \in \mathcal{I}\,(x)$ but $o \notin \mathcal{I}\,(y)$.
   (b) $y \in \mathbf{C}_D$ and $\exists C_p \in \mathbf{C}_P$ such that: $C_p \in \Phi(y)$ but $C_p \notin \Phi(x)$.
   Let $\mathcal{I}$,$\mathcal{V}$ and $o \in \mathcal{O} \subseteq \mathcal{V}$ such that $o \in \mathcal{I}\,(x)$ but $o \notin \mathcal{I}\,(C_p)$; then, given that $\mathcal{I}\,(y) \subseteq \mathcal{I}\,(C_p)$, $o \notin \mathcal{I}\,(y)$.
   (c) REF(TYP(x), TYP(y))=FALSE.
   Given that x is a coherent class, there must be TYP(x) $\leq$ TYP(C_p) $\forall C_p \in \Phi(x)$. Now by induction, there are $\mathcal{I}^*, \mathcal{V}^*, v^*$ such that $v^* \in \mathcal{I}^*\,(\mathrm{TYP}(x))$ but $v^* \notin \mathcal{I}^*\,(\mathrm{TYP}(y))$. Let o_0 be an object identifier such that $\delta(o_0) = v^*$; $\mathcal{V} = \mathcal{V}^* \cup \{o_0\}$; $\mathcal{I}$ an interpretation function over $\mathcal{V}$ such that: a) for every type s, $\mathcal{I}\,(s) \subseteq \mathcal{I}^*\,(s) \cup \{o_0\}$, b) $\forall C_p \in \Phi(x) o_0 \in \mathcal{I}\,(C_p)$. Then, from a), $v^* \in \mathcal{I}\,(\mathrm{TYP}(x))$ while the assumption b) is valid given that, $\delta(o_0) = v^* \in \mathcal{I}\,(typef x) \subseteq \mathcal{I}\,(typef C_p)\ \forall C_p \in \Phi(x)$. On the other hand, $v^* \notin \mathcal{I}\,(typef y)$, therefore, due to the definition of class interpretation, $o_0 \in \mathcal{I}\,(x)$ but $o_0 \notin \mathcal{I}\,(y)$.