Union-Types in Object-Oriented Schemas

Jan Hidders
Dept. of Math. and Comp. Science
Eindhoven University of Technology
PO box 513, 5600 MB Eindhoven
e-mail: hidders@win.tue.nl

Abstract

In this paper we investigate union-types in object oriented IQL-like schemas. These types can be used to model null values, variant types and generalization classes. They make, however, deciding equivalence and subtyping more difficult. We will show that the complexity of these two problems is co-NP-complete and present complete sets of rules for deciding both problems. The combination of union-types and multiple inheritance makes it also harder to detect typing-conflicts in a schema. We will give an algorithm for deciding this and discuss its complexity. Furthermore, we will present an algorithm for detecting schemas that define types with a bounded number of values. Finally, an algorithm will be presented that verifies whether in a schema the type of a subclass specifies options that are forbidden by its superclasses.

1 Introduction

The introduction of union-types in object-oriented schemas makes them more expressive. Usually, however, they are limited to disjoint unions [9] or labeled unions such as variant records [7]. Because of this limitation the reasoning about these types remains simple [6, 13]. We argue that *general* union-types such as used in [10, 2, 5] are a useful extension and even arise naturally in data models without union-types.

We can use them, for instance, to model optional integer fields by specifying their type as $(\texttt{null} \vee \texttt{int})$ where `null` is a special basic type with only one value viz. $null$. Moreover, we could define several kinds of nulls and let the type be $(\texttt{null}_{unkn} \vee \texttt{null}_{undef} \vee \texttt{int})$. If an object has two fields that are both optional we might want to specify that one of them has to be defined but not both. This can be done by giving the object's class the tuple-type $([a : \texttt{int}, b : \texttt{null}] \vee [a : \texttt{null}, b : \texttt{str}])$[1]. This is an example of how in general it is possible to represent *variant records*.

In data models that do not have union-types such as in [1] they arise naturally in the context of multiple inheritance. They can, for instance, be used to denote generalization classes without explicitly adding them to the schema. This can be demonstrated by the schema depicted in Figure 1. Here we see the two classes `Angler` and `Sea−Fisherman` both with the field *catch* which is, respectively, a set of `Fish` and a set of `Sea−Animals`. Since `Sea−Angler` is a subclass of `Angler` and `Sea−Fisherman` it inherits the field *catch* that must be both a set of `Fish` and a set of `Sea−Animals`. The type of the *catch* of a `Sea−Angler` is a set of $(\texttt{Mackerel} \vee \texttt{Tuna})$, or with intersection-types as a set of $(\texttt{Fish} \wedge \texttt{Sea−Animal})$ which is actually the same type. Without union-types or intersection-types this type cannot be denoted unless an extra class `Sea−Fish` is added that is the generalization of `Mackerel` and `Tuna`, and a subclass of `Fish` and `Sea−Animal`. Although this class might seem quite natural here, this approach can lead to the addition of more unnatural generalization classes.

Although union-types give us more expressiveness they also make the reasoning about types more difficult. Different type expressions can now denote the same type. For instance, the type $([a : \texttt{int}] \wedge ([b : \{\texttt{bool}\}] \vee [b : \{\texttt{str}\}]))$ is equivalent with $[a : \texttt{int}, b : \{(\texttt{bool} \vee \texttt{int})\}]$ and in the schema of Figure 1 it holds that $(\texttt{Mackerel} \vee \texttt{Fish})$ is equal

[1] It might seem more obvious to use the type $([a : \texttt{int}] \vee [b : \texttt{str}])$ but due to the subtyping semantics the two fields would not be mutually exclusive.

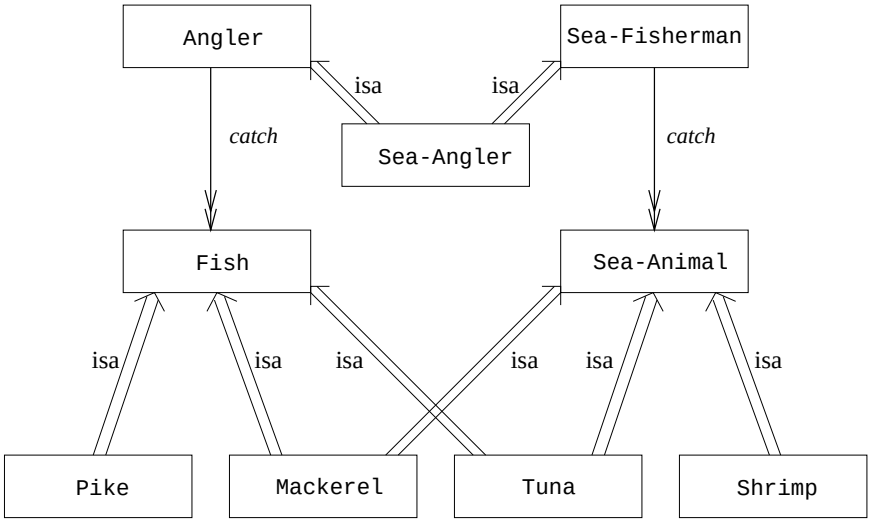

Figure 1: A schema with multiple inheritance

to `Fish`. The same problem occurs with the rules for subtyping. Although the rules for types without union-types remain the same, the rules for union-types cannot be simply defined by induction on the type. For instance, it is not *sufficient* to check that type t_1 is a subtype of type t_2 or type t_3 in order to see whether t_1 is a subtype of $(t_2 \vee t_3)$. In this paper we present complete rules for both equivalence and subtyping, and show how they provide *necessary* and *sufficient* conditions for deciding equivalence and subtyping. This results in algorithms of which we will discuss the complexity. The presented rules are similar to those presented in [3] where it is shown how a sound and complete subtyping algorithm for union types can be obtained out of some algorithm for intersection types. An example of such an algorithm for intersection types can be found in [12]. The main difference with our work is that we do not allow general functional types but only tuple-types (which might be regarded as a very limited functional type), and that we have set-types and classes as types.

The presented reasoning rules can also be used in algorithms for detecting schemas with inconsistencies and/or improbable cases. An example of an inconsistency is where the types of a class and its superclasses are not compatible. This can be checked by taking for every class the intersection of its type and the types of all the superclasses and deciding whether this type is equivalent with $\emptyset$ i.e. the empty type. If this holds then the schema is inconsistent and the class will always be empty. These problems were already discussed in [4, 11, 14] for the more conventional object-oriented data model. For data models with union- and intersection-types they were studied in [5]. The main differences between this work and ours is that we allow types of arbitrarily nested sets and tuples but we limit the classes that an object can simultaneously belong to. Another difference is that we present algorithms for detecting schemas with improbable cases that indicate modeling errors.

An example of an improbable case is a type where the value of a field can only be a bounded number of values. Let, for instance a class have two superclasses with, respectively, the types $[a : (\texttt{int} \vee \texttt{null})]$ and $[a : (\texttt{str} \vee \texttt{null})]$, then field a would always have the value $null$ with objects in the subclass. Another example would be two superclasses with, respectively, the types $[b : \{\texttt{int}\}]$ and $[b : \{\texttt{str}\}]$ where the value of field b would have to be of type $\{\emptyset\}$ in the subclass. Note that this type is not empty but contains only one value viz. $\emptyset$ (the empty set). Although these cases are not strictly wrong they might indicate a modeling error and should be detected by a CASE tool. We will present an algorithm for detecting types with a bounded number of values or with fields of tuples that have only a bounded number of possible values.

Finally, we will present an algorithm for checking whether the type of a subclass is a *proper extension* of the types of its superclasses. A type is a proper extension of another type if it does not specify options already forbidden by the other type. We say that a schema is proper iff the types specified with the classes are proper extensions of the

intersection of the types of the superclasses (not including the class itself). If a schema is not proper it is likely that it contains a modeling error. For instance, if the type of a superclass is $[a : (\texttt{int} \vee \texttt{bool})]$ and the type of the subclass is $[a : (\texttt{str} \vee \texttt{bool})]$ then the `str`-option in the subclass is superfluous and the schema is not proper. We will give a precise semantic definition of *proper extension* and provide and discuss an algorithm that decides whether a type is a proper extension of another type.

The paper is organized as follows. In Section 2 we present the data model and give the semantics of the types. In Section 3 we discuss the problems of deciding equality and subtyping for types. In Section 4 we present the algorithms for detecting inconsistent schemas, bounded types and non-proper schemas.

2 The Data Model

The data model we use is a slight extension of the IQL data model. We will only give a brief informal description.

An *instance* consists of a finite set O of objects, a function val that gives the value of every object in the instance and a function ext that gives the extension of every class. Every object in O must occur in the extension of at least one class. If an object occurs in the value of another object of O then it must also be in O.

A *value* can be either a *basic value*, an *object*, a *tuple* or a *set*. Examples of basic values are integers, strings, bitmaps and the special value $null$. Tuples are always of the form $[f_1 : v_1, \ldots, f_n : v_n]$ with $n \geq 0$, $f_1, \ldots, f_n$ distinct field names and $v_1, \ldots, v_n$ values. Sets are of the form $\{v_1, \ldots, v_m\}$ with $m \geq 0$ and $v_1, \ldots, v_m$ distinct values.

A *type* can be described by the following abstract syntax:

$$T \quad ::= \quad \emptyset \mid \mathcal{B} \mid \mathcal{C} \mid [f_1 : T, \ldots, f_n : T] \mid \{T\} \mid (T \vee T) \mid (T \wedge T)$$

Where $\emptyset$ is the empty type with no values, $\mathcal{B}$ is the set of basic types (such as `int`, `str`, `bitmap` and the special type `null`), $\mathcal{C}$ is the set of class names and $f_1, \ldots, f_n$ with $n \geq 0$ are distinct field names. Types of the form $(T \vee T)$ and $(T \wedge T)$ are, respectively, called *union-types* and *intersection types*. In the following we will let the variables τ, σ, ρ range over types, b range over basic types, c range over class names and f,g,h range over field names.

A *schema* describes the structure of a database instances. It consists of a finite set C of class names, a function $type$ that gives for every class name in C a type, and the binary relationship isa that is defined between the class names in C. If a class occurs in the type of a class in the schema then it must also be in the schema. The reflexive and transitive closure of isa is denoted as isa^*. We say that c_1 is a *direct subclass* of c_2 iff $c_1\ isa\ c_2$ and that c_1 is a *subclass* of c_2 iff $c_1\ isa^*\ c_2$. Moreover, c_1 is a *strict subclass* of c_2 if c_1 is a subclass of c_2 but c_2 is not a subclass of c_1. Since all classes have a unique name we can safely identify classes with their names.

The *semantics of a type* is given by the function $[\![\cdot]\!]_{ext}$ such that $[\![\tau]\!]_{ext}$ gives the set of values that belong to the type τ under the extension function ext. If a value is an element of the semantics of a type we also say that the value *belongs to* the type. The function is defined by induction upon the structure of the type, where $[\![\cdot]\!]^{\mathcal{B}}$ is a function that maps every basic type to a non empty countable set such that the semantics of different basic types are disjoint and $[\![\texttt{null}]\!]^{\mathcal{B}} = \{null\}$:

- $[\![\emptyset]\!]_{ext} = \emptyset$ and $[\![b]\!]_{ext} = [\![b]\!]^{\mathcal{B}}$ and $[\![c]\!]_{ext} = ext(c)$
- $[\![[f_1 : \tau_1, \ldots, f_n : \tau_n]]\!]_{ext} = \{\ [g_1 : v_1, \ldots, g_m : v_m] \mid \forall f_i \exists g_j : f_i = g_j \text{ and } v_j \in [\![\tau_i]\!]_{ext}\ \}$
- $[\![\{\tau\}]\!]_{ext} = \{\ \{v_1, \ldots, v_m\} \mid m \geq 0, \text{and } v_i \in [\![\tau]\!]_{ext}, i = 1, \ldots, m\ \}$
- $[\![(\tau_1 \vee \tau_2)]\!]_{ext} = [\![\tau_1]\!]_{ext} \cup [\![\tau_2]\!]_{ext}$ and $[\![(\tau_1 \wedge \tau_2)]\!]_{ext} = [\![\tau_1]\!]_{ext} \cap [\![\tau_2]\!]_{ext}$

We now say that *an instance belongs to a schema* iff

1. the values of objects in a class are of the type of that class,
2. the extension of a class must be a subset of the extension of all its direct superclasses, and
3. an object that belongs to two classes must also belong to some common subclass of these two classes.

Notice that condition 3 is equivalent with saying that for every object there is a unique class such that it belongs to this class and to all its superclasses and to no other class. This unique class is said to be *the class of* the object. It also follows immediately from condition 2 that the extension of a class is a subset of the extension of *all* its superclasses. A schema $\langle C, type, isa\rangle$ is *sound* if there is not a class c in C whose extension is empty in all instances of the schema.

This concludes the presentation of the data model. It must be noted here that our data model is slightly flawed because it allows values with fields which were not specified in the schema. For instance, a schema with only one class `Person` with type $[name : \texttt{str}, age : \texttt{int}]$ might have an instance consisting of an object with the value $[name : \text{“}Pete'', age : 28, sex : \text{“}male'']$. This problem is solved in the IQL data model but leads to a more complex definition of the data model. Since it does not play any role in the rest of this paper we do not present the IQL-solution here but refer the interested reader to the IQL paper [2].

3 Equality and Subtyping

In this section we will give reasoning rules for deciding subtyping and equality. We assume a fixed schema S consisting of $\langle C, type, isa\rangle$ that is sound. Under this schema a type τ_1 is a *subtype* of type τ_2, written as $\tau_1 \leq \tau_2$, if it holds for all instances $\langle O, val, ext\rangle$ of schema S that $[\![\tau_1]\!]_{ext} \subseteq [\![\tau_2]\!]_{ext}$. The types τ_1 and τ_2 are *equal*, written as $\tau_1 \equiv \tau_2$, if it holds for all instances $\langle O, val, ext\rangle$ of schema S that $[\![\tau_1]\!]_{ext} = [\![\tau_2]\!]_{ext}$. It is easy to see that $\tau_1 \equiv \tau_2$ iff $\tau_1 \leq \tau_2$ and $\tau_2 \leq \tau_1$. Therefore it will be sufficient to give the reasoning rules for subtyping only.

The reasoning rules we will give consist of two parts. The first part consists of rewrite rules that let us rewrite a type into a certain normal form. The second part consists of rules that determine whether two types in normal form are subtypes. We will begin with the rewrite rules that eliminate all the intersection-types within a type. The first set of these are those that are obvious from the interpretation of $\emptyset$, $\wedge$ and $\vee$ as, respectively, the empty set, the intersection and the union:

INT1 $(\emptyset \wedge \tau) \rightsquigarrow (\tau \wedge \emptyset) \rightsquigarrow \emptyset$

INT2 $\tau_1 \wedge (\tau_2 \vee \tau_3) \rightsquigarrow (\tau_1 \wedge \tau_2) \vee (\tau_1 \wedge \tau_3)$

INT3 $(\tau_1 \vee \tau_2) \wedge \tau_3 \rightsquigarrow (\tau_1 \wedge \tau_3) \vee (\tau_2 \wedge \tau_3)$

The second set of rewrite rules for intersection-types follows from the interpretation of basic types, set-types and tuple-types. Here we use the notation $(\tau_1 \vee \ldots \vee \tau_n)$ and $(\tau_1 \wedge \ldots \wedge \tau_m)$ for all those types that can be reduced to this by removing the brackets of, respectively, the union and the intersection-types.

INT4 $(b \wedge b) \rightsquigarrow b$

INT5 $(b_1 \wedge b_2) \rightsquigarrow \emptyset$ whenever $b_1 \neq b_2$

INT6 $(c_1 \wedge \ldots \wedge c_n) \rightsquigarrow (c'_1 \vee \ldots \vee c'_k)$ whenever $\{c'_1, \ldots, c'_k\}$ is the non empty set of maximal common subclasses of $c_1, \ldots, c_2$. A class is said to be a *maximal common subclass* of a set of classes if it is a common subclass and every strict superclass of the class is not a common subclass.

INT7 $(c_1 \wedge c_2) \rightsquigarrow \emptyset$ whenever c_1 and c_2 have no common subclass.

INT8 $([f_1 : \tau_1, \ldots, f_n : \tau_n] \wedge [g_1 : \sigma_1, \ldots, g_m : \sigma_m]) \rightsquigarrow [h_1 : \rho_1, \ldots, h_p : \rho_p]$
where $\{h_1, \ldots, h_p\} = \{f_1, \ldots, f_n\} \cup \{g_1, \ldots, g_m\}$ and
$\rho_k = \tau_i$ whenever $h_k = f_i$ and $h_k \notin \{g_1, \ldots, g_m\}$,
$\rho_k = (\tau_i \wedge \sigma_j)$ whenever $h_k = f_i = g_j$,
$\rho_k = \sigma_j$ whenever $h_k = g_j$ and $h_k \notin \{f_1, \ldots, f_n\}$.

INT9 $(\{\tau_1\} \wedge \{\tau_2\}) \rightsquigarrow \{(\tau_1 \wedge \tau_2)\}$

INT10 $(\tau \wedge \sigma) \rightsquigarrow \emptyset$ whenever τ and σ are of the form b, c, $[a_1 : \tau_1, \ldots, a_n : \tau_n]$ or $\{\tau_1\}$ but not both of the same form.

Theorem 3.1 *The rules INT1–INT10 will reduce any type to an equivalent type without intersection-types by applying them until no more rules apply.*
Proof: It is easy to verify that these rules are all sound and as long as there is an intersection-type left one of them will be applicable. Furthermore, they either remove the intersection-type entirely or push it one level down the parsing tree while leaving the entire height of the parsing tree the same. □

It should be noted here that applying these rewrite rules may lead to an exponential growth of the size of the type. This is due to the rules INT2 and INT3 that can practically double the size of the type when applied. Even if the number of intersection-types is constant the growth may still be exponential.

When the rule INT5, INT7 or INT10 can be applied there are apparently some local inconsistencies in the type. For deciding whether the type as a whole is inconsistent i.e. it is empty, we additionally need the following pair of rewrite rules:

EM1 $(\emptyset \vee \tau) \rightsquigarrow (\tau \vee \emptyset) \rightsquigarrow \tau$

EM2 $[\ldots, f_i : \emptyset, \ldots] \rightsquigarrow \emptyset$

Theorem 3.2 *The rules INT1–INT10, EM1 and EM2 reduce a type τ to $\emptyset$ iff $\tau \equiv \emptyset$*
Proof: The only-if part follows form the soundness of the rewrite rules. The if part can be easily proved by induction upon the structure of the type using the fact that a type without intersection-types can only be empty (under a sound schema) if it is either the empty type, a tuple-type with an empty field or a union-type consisting of two empty types. □

Since the rewrite rules cause an exponential blow-up of the type this algorithm for deciding emptyness is also exponential. That it will be difficult to find an algorithm that is not exponential in time follows from the following theorem.

Theorem 3.3 *Deciding whether a type is empty is co-NP-complete.*
Proof: First we show that the problem is in co-NP by demonstrating that there is an NP algorithm for deciding the inverse problem. This algorithm consists off guessing a *constituent* of the type that is not empty. A constituent of a type is obtained by replacing all subexpressions of the form $(\tau_1 \vee \tau_2)$ not nested in a set-type with either τ_1 or τ_2. It holds that every type is equivalent with the union of its constituents. Thus a type is not empty iff one of its constituents is not empty. We can now reduce the part of the type that is not nested in a set-type with the INT and EM rules in polynomial time because the rules INT2 and INT3 can never become applicable. Because set-types are never empty (they always contain at least the empty set) it holds that the type is now reduced to $\emptyset$ iff it is empty. Therefore it is decidable in polynomial time whether a constituent is empty or not.

The hardness result is shown by reducing the problem of checking if a CNF formula ϕ is unsatisfiable, known to be co-NP-complete [8]. This can be done by translating the formula ϕ to a type that is empty iff ϕ is not satisfiable. For every variable x_i in ϕ we choose a different field name f_i. Then we transform ϕ by to a type τ by replacing every literal x_i with $[f_i : []]$ and every literal $\neg x_i$ with $[f_i : \{[]\}]$. If there is a value of type τ then it defines an assignment that satisfies ϕ by letting x_i be *true* or *false* if the value of every field f_i is, respectively, a tuple or a set. Conversely, we can transform any assignment that satisfies ϕ to a value that belongs to τ by letting every field f be either the value $[]$ or $\{\}$ if x_i is assigned to, respectively, *true* or *false*. Thus τ is empty iff ϕ is satisfiable. □

The final rewrite rule splits tuple-types that have a union-type as the type of one of their fields.

SP $[f_1 : \tau_1, \ldots, f_i : (\tau_i \vee \tau_i'), \ldots, f_n : \tau_n] \rightsquigarrow ([f_1 : \tau_1, \ldots, f_i : \tau_i, \ldots, f_n : \tau_n] \vee [f_1 : \tau_1, \ldots, f_i : \tau_i', \ldots, f_n : \tau_n])$

We say that if no one of the rules INT1–INT10, EM1, EM2 or SP applies to a type then it is in *split normal form*. It is easy to see that a type is in split normal form iff it contains no intersection-types, $\emptyset$ is not an argument of a union-type, and the type of a tuple field is neither $\emptyset$ nor a union-type. A type is in *partial split normal form* if this only holds for those parts of the type that are not nested inside a set-type. This normal form is interesting because a type in partial split normal form can not be split into two semantically strictly smaller types unless it is already a union-type.

$$\textbf{SR1}\ \frac{}{\emptyset \sqsubseteq \tau} \qquad \textbf{SR2}\ \frac{}{b \sqsubseteq b} \qquad \textbf{SR3}\ \frac{c_1\ isa^*\ c_2}{c_1 \sqsubseteq c_2}$$

$$\textbf{SR4}\ \frac{\forall g_j \exists f_i : f_i = g_j \wedge \tau_i \sqsubseteq \sigma_j}{[f_1 : \tau_1, \ldots, f_n : \tau_n] \sqsubseteq [g_1 : \sigma_1, \ldots, g_m : \sigma_m]} \qquad \textbf{SR5}\ \frac{\tau \sqsubseteq \sigma_1 \text{ and } \tau \sqsubseteq \sigma_2}{\tau \sqsubseteq (\sigma_1 \wedge \sigma_2)}$$

$$\textbf{SR6}\ \frac{\tau_1 \sqsubseteq \tau_2}{\{\tau_1\} \sqsubseteq \{\tau_2\}} \qquad \textbf{SR7}\ \frac{\tau_1 \sqsubseteq \sigma \text{ and } \tau_2 \sqsubseteq \sigma}{(\tau_1 \vee \tau_2) \sqsubseteq \sigma} \qquad \textbf{SR8}\ \frac{\tau \sqsubseteq \sigma_1 \text{ or } \tau \sqsubseteq \sigma_2}{\tau \sqsubseteq (\sigma_1 \vee \sigma_2)}$$

Figure 2: Rules for the syntactical subtyping relationship.

Theorem 3.4 *If a type τ is in partial split normal form and not a union-type then there are no two types σ_1 and σ_2 semantically strictly smaller than τ such that $\tau \equiv (\sigma_1 \vee \sigma_2)$.*
Proof: (Sketch) It can be proved with induction on the structure of the type that it can not hold that $\tau \equiv (\tau_1 \vee \ldots \vee \tau_n)$ when $\tau_1, \ldots, \tau_n$ are all in partial split normal form, not union-types and strictly smaller than τ. This proof uses the facts that there is no type of the form $(\rho_1 \vee \ldots \vee \rho_m)$ that contains *all* values, and that for every class in the schema there is an instance where the class contains objects not in one of its subclasses. From this proof it follows that τ can not be split in σ_1 and σ_2 because otherwise we could rewrite $(\sigma_1 \vee \sigma_2)$ to a type of the form $(\tau_1 \vee \ldots \vee \tau_n)$ with $\tau_1, \ldots, \tau_n$ all in partial split normal form, not union-types and strictly smaller than τ. □

Thanks to this property of types in split normal form we can now give a simple definition of the *syntactical subtyping relationship* $\sqsubseteq$ that is a binary relationship between types and defined by the rules in Figure 2.

Theorem 3.5 *For every type τ and σ with τ in split normal form it holds that $\tau \sqsubseteq \sigma$ iff $\tau \leq \sigma$.*
Proof: The soundness of the rules is easily verified and holds even for types not in split normal form. The completeness is proved with induction upon the structure of τ and σ by showing firstly that if $\tau \leq \sigma$ then they are in the form of one of the conclusions of the rules, and secondly that in all the rules the premises are *necessary* conditions except for rule SR8 where the premise is only necessary if τ is not a union-type and in split normal form. This last fact follows easily from Theorem 3.4. Thus the premise of SR8 is a necessary condition if SR7 is not applicable. □

Although the rule SP is an extra source of exponential growth a type grows only single exponential when rewritten to split normal form. That it will be hard to find an algorithm for types without intersection-types that is not exponential in time is shown by the following theorem.

Theorem 3.6 *Deciding whether $\tau \leq \sigma$ is co-NP-complete.*
Proof: Hardness is easy because $\tau \leq \emptyset$ iff $\tau \equiv \emptyset$ which was proven co-NP-hard in Theorem 3.3.

We show that it is in co-NP by construction of a non-deterministic algorithm in polynomial time. The algorithm has to be both polynomial in the size of the schema and the types τ and σ. First it guesses a constituent and then it reduces intersection-types except those between classes and those nested in a set type. Then it applies INT6 to the largest possible intersections of classes not nested in a set-type and guesses a (sub)constituent by choosing one of the common subclasses. The now remaining intersections that are not nested in a set-type are all reducible to $\emptyset$. This is because every such intersection is either between a class and another intersection or between a class and a type that is neither a class nor a union-type. Finally, empty types are removed using the EM rules. Note that $\tau \leq \sigma$ iff every one of its (sub)constituents is a subtype of σ.

The type τ' that is now constructed is in partial split normal form. We can now use the rules of the $\sqsubseteq$-relationship to determine if this type is a subtype of σ. If it is necessary for this procedure to decide whether $\tau'' \sqsubseteq \sigma'$ where τ'' is a subexpression of τ' inside a set-type then we can repeat the same procedure for τ''. □

Due to rule SP the algorithm even remains exponential if τ does not contain any intersection-types. It can, however, be shown that even this subproblem is already intractable.

Theorem 3.7 *Deciding whether $\tau \leq \sigma$ where τ and σ contain no intersection types, is co-NP-hard.*
Proof: This is shown by reducing the problem of checking if a CNF formula ϕ is unsatisfiable. We will construct the types τ and σ such that they are polynomial in the size of ϕ and that $\tau \leq \sigma$ iff there is no assignment satisfying ϕ.

For the propositional variables $x_1, \ldots, x_n$ in the formula ϕ we construct the types $\pi_1, \ldots, \pi_{2n}$ as follows. Let $g_1, \ldots, g_m$ be distinct field names with $m = \log(2n)$, then $\pi_i = [g_1 : \nu_1, \ldots, g_m : \nu_m]$ with $\nu_j = []$ if the j'th bit of the binary representation of i is 1, and $\nu_j = \{[]\}$ if it is 0. Note that the semantics of all π_i's are pairwise disjoint and that their size is logarithmic in the number of variables in ϕ.

The type τ is constructed as follows. Assume that the formula ϕ is of the form $\psi_1 \wedge \ldots \wedge \psi_k$. Then $\tau = [f_1 : \rho_1, \ldots, f_k : \rho_k]$ with $f_1, \ldots, f_k$ distinct field names and where ρ_i is obtained from ψ_i by replacing the literals x_i with π_i and $\neg x_i$ with π_{i+n}.

We construct σ as the union of all tuple-types with exactly fields $f_1, \ldots, f_k$ of which two fields contain π_i and π_{i+n}, respectively, for some x_i and the other fields contain the type $(\pi_1 \vee \ldots \vee \pi_{2n})$. Note that the number of such tuple-types is $nk(k-1)$ and that the size of σ is therefore polynomial in the size of ϕ.

If we assume that an assignment can assign a variable to both *true* and *false* then it holds that every constituent of τ corresponds with a certain assignment of the variables satisfying ϕ and vice versa. All the constituents of σ correspond with inconsistent assignments (i.e. some x_i is bound to both *true* and *false*) and vice versa. So it is clear that τ is a subtype of σ iff all the assignments satisfying ϕ are inconsistent. □

4 Schema Assessment

4.1 Sound Schemas

The soundness of a schema was defined as the property that no class may be empty in *all* instances of the schema. It is clear that this is a desirable property for a schema and that unsound schemas indicate a modeling error. Therefore a syntactical formulation of soundness is required so that it can be checked algorithmically. An example of an unsound schema is the schema $\langle C, type, isa \rangle$ with:

$C = \{\texttt{Secretary}, \texttt{Employee}\}$,
$type(\texttt{Employee}) = [name : \texttt{str}, address : \texttt{str}, birth\_date : \texttt{int}]$,
$type(\texttt{Secretary}) = [birth\_date : \texttt{str}]$,
$(\texttt{Secretary}\ isa\ \texttt{Employee})$

Here the class `Secretary` is empty in every instance. This is because every secretary is also an employee and therefore the value of a secretary must be both of type $[birth\_date : \texttt{str}]$ and $[name : \texttt{str}, address : \texttt{str}, birth\_date : \texttt{int}]$, which is impossible. The value of an object of a certain class must always be of all the types of the superclasses (including the class itself). This is equivalent with saying that it should be of the type that is the intersection of the types of all the superclasses. This type is said to be the *full type* of that class. In the example above the full type of `Secretary` is $([birth\_date : \texttt{str}] \wedge [name : \texttt{str}, address : \texttt{str}, birth\_date : \texttt{int}])$. It is clear that if the full type of a class in a schema denotes the empty type then the schema is not sound. Since the reverse can also be shown to hold we obtain the following syntactical characterization of the soundness of a schema.

Theorem 4.1 *A schema is sound iff no full type of any class in the schema is reduced to $\emptyset$ by the rules INT1–INT10, EM1 and EM2.*
Proof: The only-if part follows from the soundness of the rules and the observation that the objects in a class must have a value that is of the full type of that class. The if-part is proved by the construction of an instance with for every class an object of that class i.e. this object belongs only to this class and its superclasses. In such an instance a type within the schema not reduced to $\emptyset$ by the rules 1–12 has a non-empty semantics. Therefore we can choose a value for every object in every class since the semantics of the full types is not empty. □

Since our algorithm for deciding emptyness is exponential in time this will also hold for the algorithm deciding soundness. This remains even true if the types in the schema do not contain intersection-types and the number of

classes remains constant. It is, however, also easy to see how the proof of Theorem 3.3 might be adapted to prove that checking soundness is co-NP-hard.

If there are no union-types in the schema then the algorithm is polynomial. This special case was already mentioned in [5] as solvable in polynomial time. On the other hand it is easy to see that deciding soundness is NP-complete because deciding emptyness of types is already co-NP-complete.

It is interesting to compare our work with that in [5] where soundness is stated to be EXPTIME-hard for a different kind of data model. The main difference between our and their data model is that they allow objects to belong arbitrarily to several classes at once. Furthermore, they limit the types of classes to the form $[f_1 : \{\sigma_1\}, \ldots, f_n : \{\sigma_n\}]$ with σ_i being a type formed of union-types, intersection-types, negation-types and classes. Negation-types are written as $\neg\sigma$ and contain all the objects that are not of type σ. Also it is possible to specify a type such as σ_i above as the superclass of a class. Finally, they allow to- and from-cardinalities to be specified with every field. We conjecture that the fact that objects can simultaneously belong to any set of classes explains the main difference in complexity.

If we limit the types in our data model in the same way but maintain that objects can only belong to several classes if they belong to some common subclass, the soundness becomes decidable in polynomial time. This follows from the fact that the emptyness of types consisting of union-types, intersection-types and classes can be decided in polynomial time by checking if there is some class that is a subtype of this type. If there is not such a class then the type is empty. Note that we did not add negation-types to our data model in this case. If we also want objects to be able to belong simultaneously to an arbitrary set of classes then we can simulate this by extending the schema with a class for every subset of the classes in the schema. Evidently this would cause the schema to grow exponentially.

4.2 Schemas with Bounded Types

In the previous section we presented an algorithm for detecting typing conflicts between subclasses and superclasses or between common superclasses. Sometimes, however, typing conflicts do not render the full type of a class completely empty. An example would be a class with two superclasses having, respectively, the types $[a : (\texttt{int} \vee \texttt{null})$ and $[a : (\texttt{string} \vee \texttt{null})]$. The full type of the class would then reduce to $[a : \texttt{null}]$ and not be empty although we might speak of a design error. Another example is where the two superclasses have the types $[a : \{\texttt{int}\}]$ and $[a : \{\texttt{string}\}]$. Then the full type would reduce to $[a : \{\emptyset\}]$ which is also not empty ($[\![[a : \{\emptyset\}]]\!]_{ext} = \{[a : \emptyset]\}$) but nevertheless might indicate a design error. Finally, if the two superclasses have the types $\{\{\texttt{int}\}\}$ and $\{\{\texttt{string}\}\}$ then the full type would reduce to $\{\{\emptyset\}\}$. This type contains even two values ($\{\emptyset\}$ and $\emptyset$) but still might indicate a design error.

These examples demonstrate that when the full type or a field within the type is bounded i.e. has only a bounded number of values, then this might indicate a modeling error. Therefore we present an algorithm that determines the potential[2] cardinality of a type. This algorithm can also be used to check whether certain fields of tuples (as opposed to the complete type) are bounded. How this is done will be indicated later.

First we reduce the full type with the rules INT1–INT10 so it contains no more intersection-types. Then we determine its cardinality with the function $\lceil\cdot\rceil$ such that $\lceil\tau\rceil$ gives the maximum cardinality of $[\![\tau]\!]_{ext}$. If there is no maximum cardinality then the result is defined as ∞. The function is defined by the following equations:

BT1 $\lceil\emptyset\rceil = 0$,

BT2 $\lceil b\rceil = 1$ if $b = \texttt{null}$ and $\lceil b\rceil = \infty$ otherwise,

BT3 $\lceil c\rceil = \infty$

BT4 $\lceil[f_1 : \tau_1, \ldots, f_n : \tau_n]\rceil = \lceil\tau_1\rceil \times \ldots \times \lceil\tau_n\rceil$ where $\infty \times x = x \times \infty = \infty$

BT5 $\lceil\{\tau\}\rceil = 2^{\lceil\tau\rceil}$ where $2^\infty = \infty$

BT6 $\lceil(\tau_1 \vee \tau_2)\rceil = \lceil\tau_1\rceil + \lceil\tau_2\rceil - \lceil\tau_3\rceil$ where τ_3 is $(\tau_1 \wedge \tau_2)$ reduced by rules INT1–INT10, and $\infty + x = x + \infty = \infty - x = \infty$

[2] We can determine only the *potential* cardinality since it may depend upon the particular extension function how many objects of a certain class there are.

It is easy to see that this algorithm can become already exponential in its first step where the intersection-types are reduced. This is then even worsened by rule **BT6**. If, however, we only need to know *whether* the type is bounded then this rule can be simplified by omitting $-\lceil\tau_3\rceil$ and the total algorithm becomes single exponential.

Theorem 4.2 *Deciding boundedness is co-NP-complete.*
Proof: The hardness follows easy from Theorem 3.3 since the type $[a : \mathtt{int}, b : \tau]$ is bounded iff τ is empty.

We show that it is in co-NP by construction of a non-deterministic algorithm in polynomial time deciding the inverse problem, i.e., *unboundedness*. The algorithm has to be both polynomial in the size of the schema and the type. First it guesses a constituent and then it reduces intersection-types except those between classes and those nested in a set type. Then it applies INT6 to the largest possible intersections of classes not nested in a set-type and guesses a (sub)constituent by choosing one of the common subclasses. The now remaining intersections that are not nested in a set-type are all reducible to $\emptyset$. This is because every such intersection is either between a class and another intersection or between a class and a type that is neither a class nor a union-type. Finally, empty types are removed using the EM rules. Note that τ is unbounded iff one of its (sub)constituents unbounded.

For deciding the boundedness of the obtained constituent we can now use rule BT5 and apply the same procedure recursively to the parts nested within set-types. □

If the original type does not contain any union-types then the first step will also be polynomial and therefore also the total algorithm. If the original contains no intersection-types then it is easy to see that deciding boundedness is also polynomial since the first step will do nothing.

Instead of checking whether a whole type is bounded we also would like to check whether there are any ***fields*** in the type that are bounded. An obvious way to do this would be checking all the fields of the subexpressions that are tuple-types. This, however is too crude since the type $([a : \mathtt{null}, b : \mathtt{int}] \vee [a : \mathtt{int}, b : \mathtt{null}])$ would then be considered suspect while it does not really limit the value of either field a or b to a bounded number of values. Therefore we limit this approach to fields of tuple-types that are not arguments of union types. The tuple-types that *are* arguments of union-types are treated in the following way. For every field name f we look for the maximal subexpression of the form $([\ldots f : \sigma_1 \ldots] \vee \ldots \vee [\ldots f : \sigma_n \ldots])$ and determine whether $(\sigma_1 \vee \ldots \vee \sigma_n)$ is bounded. In the example above this would amount to checking for field a whether $\lceil(\mathtt{null} \vee \mathtt{int})\rceil = 1$ and for field b whether $\lceil(\mathtt{int} \vee \mathtt{null})\rceil = 1$.

4.3 Types with Redundant Options

The purpose of the type that is specified with a class in a schema is to give the specialization of the types of the superclasses. This type should ideally give only the *extra* restrictions to those that are already stated by the types of the superclasses. Moreover, it should not specify options that are already forbidden by these types. For instance, if the type of a superclass is $[a : (\mathtt{int} \vee \mathtt{bool})]$ and the type of the subclass is $[a : (\mathtt{str} \vee \mathtt{bool})]$ then the `bool`-option in the subclass is superfluous and indicates a modeling error. A more complicated example would be the schema $\langle C, type, isa\rangle$ with:

$$
\begin{array}{l}
C = \{\mathtt{Flight}, \mathtt{International{-}Flight}\},\\
type(\mathtt{Flight}) = [\, load : (\{\mathtt{Person}\} \vee \{\mathtt{Goods}\})\}]\\
type(\mathtt{International{-}Flight}) = [\, load : \{(\mathtt{Person} \vee \mathtt{Goods})\},\\
\qquad\qquad\qquad\qquad int\_flight\_nr : \mathtt{str}]\\
(\mathtt{International{-}Flight}\ isa\ \mathtt{Flight})
\end{array}
$$

The type of `International−Flight` allows the load to be a mixture of persons and goods. The type of superclass `Flight`, however, allows only loads that consist either of only persons or only goods. The type of the field $load$ in the subclass may therefore be considered somewhat misleading; it seems to imply that international flights can carry mixed loads. It would have been better if the field $load$ was not specified in the type of `International−Flight` at all. Informally, we would like it to hold for the type of a class that all the values of that type that have *only the specified fields* can be extended to a value that belongs to the types of the superclass by only adding extra fields. This constraint captures the intuition that if a certain type is specied for a certain field then that field should be able to become all the values of that type and not just a smaller subset. It is clear that the example above does not fulfill the condition

PR1 $\dfrac{}{\emptyset \trianglelefteq \tau}$ **PR2** $\dfrac{}{b \trianglelefteq b}$ **PR3** $\dfrac{c_1\ isa^*\ c_2}{c_1 \trianglelefteq c_2}$

PR4 $\dfrac{\forall g_j, f_i : f_i = g_j \Rightarrow \tau_i \trianglelefteq \sigma_j}{[f_1 : \tau_1, \ldots, f_n : \tau_n] \trianglelefteq [g_1 : \sigma_1, \ldots, g_m : \sigma_m]}$ **PR5** $\dfrac{\tau \trianglelefteq \sigma_1 \text{ and } \tau \trianglelefteq \sigma_2}{\tau \trianglelefteq (\sigma_1 \wedge \sigma_2)}$

PR6 $\dfrac{\tau_1 \trianglelefteq \tau_2}{\{\tau_1\} \trianglelefteq \{\tau_2\}}$ **PR7** $\dfrac{\tau_1 \trianglelefteq \sigma \text{ and } \tau_2 \trianglelefteq \sigma}{(\tau_1 \vee \tau_2) \trianglelefteq \sigma}$ **PR8** $\dfrac{\tau \trianglelefteq \sigma_1 \text{ or } \tau \trianglelefteq \sigma_2}{\tau \trianglelefteq (\sigma_1 \vee \sigma_2)}$

Figure 3: Rules for the syntactical proper extension relationship.

because the value $[load : \{person_{1253}, mail\_bag_{15}\}, int\_flight\_nr : \text{“}KL-384'']$ does not belong to the type of `Flight`. If a schema does fulfill this condition then we say it is a *proper schema*. Notice that the fact that we only consider *specified* fields is important because otherwise the constraint would amount to demanding that the type of a class is a subtype of the types of the superclasses. This would make it necessary for the modeler to specify explicitly the full type with every class.

Before we proceed with the formal definition of a *proper schema* we will formally define what it means for a value to be of a certain type and have only the specified fields. The set of these values is called the *minimal semantics of a type* and is given by the function $[\![\cdot]\!]^-_{ext}$ which is defined by the same rules as $[\![\cdot]\!]_{ext}$ (with $[\![\cdot]\!]_{ext}$ replaced by $[\![\cdot]\!]^-_{ext}$) except for tuple-types and intersection-types:

- $[\![[f_1 : \tau_1, \ldots, f_n : \tau_n]]\!]^-_{ext} = \{\ [f_1 : v_1, \ldots, f_n : v_n] \mid \forall f_i : v_j \in [\![\tau_i]\!]^-_{ext}\ \}$
- $[\![(\tau_1 \wedge \tau_2)]\!]^-_{ext} = [\![\sigma]\!]^-_{ext}$ where σ is $(\tau_1 \wedge \tau_2)$ reduced by rules INT1–INT10.

If a value is in the minimal semantics of a type we say that it is a *minimal value of* that type. It is easy to see that it holds for every type τ that $[\![\tau]\!]^-_{ext} \subseteq [\![\tau]\!]_{ext}$. Notice also that every value in $[\![\tau]\!]_{ext}$ can be trimmed down to a value in $[\![\tau]\!]^-_{ext}$ by omitting fields.

We now define a *proper schema* as a schema for which it holds for every class that all the minimal values of the type of that class can be extended to a value that is of all the types of the superclasses by only adding extra fields. On the level of individual types we say that type τ is a *proper extension of* type σ, written as $\tau \preceq \sigma$, if all the minimal values of τ can be extended to a value of σ by adding extra fields. It may be clear that a value can be extended to a value that is of the types $\tau_1, \ldots, \tau_k$ iff it can be extended to a value of the type $(\tau_1 \vee \ldots \vee \tau_k)$. Therefore a schema is proper iff it holds for all classes with type σ and $\tau_1, \ldots, \tau_k$ as the types of the superclasses (not including the type of the class itself) that $\sigma \preceq (\tau_1 \vee \ldots \vee \tau_k)$.

Deciding whether it holds that $\tau \preceq \sigma$ can be done by reducing τ to split normal form with rules INT1–INT10, EM1, EM2 and SP and then using the binary relationship $\trianglelefteq$ between types defined by the rules in Figure 3.

Theorem 4.3 *For every type τ and σ with τ in split normal form in this schema in split normal form it holds that $\tau \trianglelefteq \sigma$ iff $\tau \preceq \sigma$.*

Proof: The soundness of the rules is easily verified and holds even for types not in split normal form. The completeness is proved by showing firstly that if $\tau \preceq \sigma$ then they are in the form of one of the conclusions of the rules, and secondly that in all the rules the conditions are necessary conditions except for rule PR8 where the condition is only necessary if τ is not a union-type and in split normal form. Note that if none of the rule PR7 is applicable then the left type is not a union-type. □

The complexity of this algorithm is similar to the one we introduced for deciding subtyping. Furthermore, the problem can in mostly the same way as for subtyping be proved to be co-NP-complete.

5 Conclusions

We have discussed the problems of reasoning about equality and subtyping in IQL-like schemas containing union-types and intersection-types. It was shown that these problems are co-NP-complete but that there are relatively simple sets of rules that can be used to decide them. The presented rules are similar to those presented in [3] where it is shown how a sound and complete subtyping algorithm for union types can be obtained out of some algorithm for intersection types. The main differences with our work is that we do not allow general functional types but only tuple-types and that we have set-types and classes as types.

We used these rules to detect inconsistent schemas and showed that this problem is also co-NP-complete. These results are comparable to those in [5] where the same problem was studied for schemas that allowed objects to belong simultaneously to an arbitrary set of classes but limited their values to relatively simple tuples. Finally, we also have used the reasoning rules as a starting point for algorithms detecting situations in schemas that might indicate a modeling error.

Acknowledgments: I would like to thank Jan Paredaens for encouraging me to write this paper and helping me writing it. Furthermore, I would like to thank Val Tannen and Peter Buneman for pointing me to related work and, finally, the anonymous referees for their helpful remarks which helped me to improve this paper.

References

[1] S. Abiteboul and R. Hull. IFO: A formal semantic database model. *ACM Transactions on Database Systems*, 12(4):525–565, December 1987.

[2] S. Abiteboul and P.C. Kanellakis. Object identity as a query language primitive. In J. Clifford, B. Lindsay, and D. Maier, editors, *Proc. of the 1989 ACM SIGMOD Int'l Conf. on Management of Data*, number 18:2 in SIGMOD Record, pages 159–173. ACM Press, 1989.

[3] F. Barbanera, M. Dezani-Ciancaglini, and U. de'Liguoro. Intersection and union types: Syntax and semantics. *Information and Computation*, 119(2):202–230, 1995.

[4] K. Brathwaite. *Object-Oriented Database Design: Concepts and Application*. Academic Press, Inc., 1993.

[5] D. Calvanese and M. Lenzerini. Making object-oriented schemas more expressive. In *Proc. of the 13th ACM Symp. on Principles of Database Systems*, pages 243–254, 1994.

[6] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

[7] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In *Proc. of the 1994 ACM SIGMOD Int'l Conf. on Management of Data*, pages 313–324, 1994.

[8] M.R. Garey and D.S. Johnson. *Computers and Intractability – A guide to NP-completeness*. W.H. Freeman and Company, San Francisco, 1979.

[9] G.M. Kuper and M.Y. Vardi. The logical data model. *ACM Transactions on Database Systems*, 18(3):379–413, September 1993.

[10] C. Lécluse and P. Richard. Modeling complex structures in object-oriented databases. In *Proc. of the 8th ACM Symp. on Principles of Database Systems*, pages 360–368, 1989.

[11] T. W. Ling and P. K. Teo. Inheritance conflicts in object-oriented systems. In *Proc. of the 4th Int'l Conf. on Database and Expert Systems Applications*, number 720 in Lecture Notes in Computer Science, pages 189–200. Springer-Verlag, 1993.

[12] B.C. Pierce. A decision procedure for the subtype relation on intersection types with bounded variables. Technical Report CMU-CS-89-169, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3890, August 1989. from: `http://www.cl.cam.ac.uk/users/bcp1000/ftp/index.html`.

[13] D. Remy. Typechecking records and variants in a natural extension of ML. In *Proc. of ACM symp. on Principles of Programming*, pages 242–249, 1989.

[14] K.D. Schewe, B. Thalheim, and I. Wetzel. Foundations of object-oriented database concepts. Technical report, University of Hamburg, 1992.