# The Type-Consistency Problem for Queries in Object-Oriented Databases

Yasunori ISHIHARA Shougo SHIMIZU Hiroyuki SEKI Minoru ITO

Graduate School of Information Science
Nara Institute of Science and Technology
8916–5, Takayama, Ikoma, Nara 630-0101 JAPAN

**Abstract**

Method invocation mechanism is one of the essential features in object-oriented programming languages. This mechanism contributes to data encapsulation and code reuse, but there is a risk of a run-time type error. In the case of object-oriented databases (OODBs), a run-time error causes rollback. Therefore, it is desirable to ensure that a given OODB schema is consistent, i.e., no run-time type error occurs during the execution of queries under any database instance of the OODB schema.

This paper discusses the computational complexity of the type-consistency problem. As a model of OODB schemas, we adopt update schemas introduced by Hull et al., which have all of the basic features of OODBs such as class hierarchy, inheritance, complex objects, and so on. For several subclasses of update schemas, the complexity of the type-consistency problem is presented. Importantly, it turns out that non-flatness of the class hierarchy, recursion in the queries, and update operations in the queries each make the problem difficult.

## 1. Introduction

Among many features of object-oriented programming languages (OOPLs), method invocation (or message passing) mechanism is an essential one. It is based on method name overloading and late binding by method inheritance along the class hierarchy. For a method name $m$, different classes may have different definitions (codes, implementations) of $m$. When $m$ is applied to an object $o$, one of the definitions of $m$ is selected depending on the class to which $o$ belongs, and is bound to $m$ in run-time (late binding or dynamic binding). This mechanism is important for data encapsulation and code reuse, but there is a risk of a run-time type error. For example, when a method $m$ is invoked, the definition of $m$ to be bound may not be uniquely determined. Particularly with queries in object-oriented databases (OODBs), a run-time error causes rollback, i.e., all the modification up to the error must be cancelled.

In this paper, we discuss the computational complexity of the type-consistency problem for queries in OODBs. A database schema **S** is said to be *consistent* if no type error occurs during the execution of any method under any database instance, i.e.,

1. for every method invocation $m$, the definition of $m$ to be bound is uniquely determined through the class hierarchy with inheritance; and

2. no attribute-value update violates any type declaration given by **S**.

In order to check type-consistency, it is usually necessary to perform type inference, i.e., to examine whether for each class $c$ and program construct $x$ such as a variable in method implementation bodies, the value of $x$ can be an object of class $c$ or not. It is quite advantageous for a given database schema to be consistent. First, since it is ensured at compile-time that no type error occurs under any database instance, run-time type check can be omitted. Another advantage is an application to method-based authorization checking [5], [7], [16].

As a model of OODB schemas, we adopt *update schemas* introduced by [11]. Update schemas have all of the basic features of OODBs, such as class hierarchy, inheritance, complex objects, and so on. Method implementations are based on a procedural OOPL model. Therefore, updating database instances is simply modeled as assignment of objects or basic values to attributes of objects. In [11], it is shown that the type-consistency problem for update schemas

is undecidable. In [16], a subclass of update schemas, called *non-branching update schemas*, is introduced. It is shown that consistency for a given non-branching update schema is solvable in polynomial time provided that all the database instances are acyclic.

The aim of this paper is to investigate the computational complexity of the type-consistency problem for several subclasses of OODB schemas. We focus on the following three factors and show that each of them creates difficulty in the problem (see also Fig. 1):

1. *Non-flatness of the class hierarchy* (Section 3.1). Define the *height* of the class hierarchy as the maximum length of a path in the hierarchy. If the height is zero, then all classes are completely separated and there is no superclass-subclass relation at all. For such a "flat" database schema, consistency is solvable in polynomial time. However, consistency for a non-flat schema is undecidable even if it is *retrieval* (i.e., no method definition in the schema contains any update operation) and the height of the class hierarchy is bounded by one.

2. *Recursion* (Section 3.2). Consistency for a recursion-free schema is coNEXPTIME-complete, while consistency for a schema with recursion is undecidable even if it is *terminating* (i.e., the execution of every method terminates under every database instance) and the height of the class hierarchy is bounded by one.

3. *Update operations* (Section 3.3). As stated above, consistency for a terminating (resp. recursion-free) schema with update operations is undecidable (resp. coNEXPTIME-complete), even if the height of the class hierarchy is bounded by one. On the other hand, consistency for a terminating retrieval schema is solvable in polynomial time. Thus, update operations make the consistency problem difficult when the schema satisfies the termination property.

The model adopted in this paper requires the following three conditions. First, schemas should be monadic (i.e., every method in a schema should have arity one). Even if the arity is not bounded, consistency is expected to be still decidable for a flat schema, a recursion-free schema, and a terminating retrieval schema respectively. That is, in our conjecture, arity does

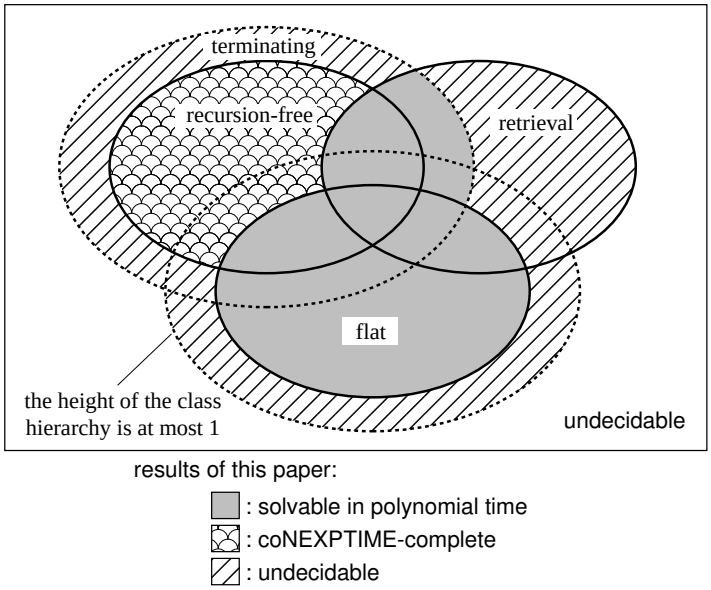



Fig. 1 Complexity of the Type-Consistency Problem.

not affect the decidability of consistency as long as we consider only the subclasses of schemas stated above. Secondly, there should be no program constructs such as conditional branch and while statement. However, using update operations, if-then statements can be simulated (see Example 3). Thirdly, the class hierarchy should be a forest (i.e., multiple inheritance is excluded). However, the results in this paper remain valid if an appropriate mechanism for multiple inheritance is incorporated into the model. That is, the third condition is merely for simplicity.

There has been much research on the type-consistency problem for OOPLs. As a pioneer work, Abiteboul et al. [2], [3] introduced *method schemas* and studied the complexity of the type-consistency problem for a number of subclasses. In method schemas, each method is allowed to have more than one argument. However, method schemas cannot represent updates of database instance since their method implementations are based on a functional OOPL model. The followings are some of the main results and open problems of [2]:

1. Consistency is undecidable for a general method schema;

2. It is open for a method schema with methods of arity two;

3. It is coNP-complete for a recursion-free method schema provided that the arity of each method is bounded by a constant; and

4. It is solvable in polynomial time for a monadic method schema.

Retrieval schemas of ours are a proper subclass of general method schemas and a proper superclass of monadic method schemas. Moreover, retrieval schemas are incomparable to method schemas with methods of arity two, and their intersection is not empty. In this paper, we provide a proof of undecidability for a retrieval schema which belongs to the intersection. That is, the open problem 2 above is shown to be undecidable. In [17], an optimal incremental algorithm for the consistency checking of a recursion-free method schema is presented. In [1], the complexity of type-consistency (and also the expressive power) for both update and method schemas is summarized.

As already stated, type inference is closely related to type-consistency. In [14], a type inference algorithm for a procedural OOPL is proposed. For each expression $e$ of a program, a type variable $[\![e]\!]$ that denotes the type of $e$ is introduced, and a sufficient condition for type-consistency can be examined by computing the least solution of the equations that denote the relations among these type variables (also see [13] and [15]).

Our OOPL model is *untyped* in the sense that each variable has no type declaration. In contrast, type-consistency for *typed* OOPLs have been discussed in several articles [4], [6], [8]. Since the language is typed in these articles, it can be assumed that we know in advance the class to which the returned objects should belong for every method implementation body. Then the consistency problem is simply to determine whether each method satisfies conditions such as covariance and contravariance. Therefore, for typed OOPLs, behavioral analysis of each method implementation body is unnecessary. Type systems for OOPLs have also been extensively studied [9], [10]. For example, in [10], an elegant type system is proposed that relaxes contravariance restriction. However, computational complexity of the type-consistency problem has scarcely been studied in these articles.

The remainder of the paper is organized as follows. In Section 2, we define database schemas and their instances, and show some examples. In Section 3, we show the computational

complexity of the type-consistency problem for the subclasses of database schemas mentioned above. Lastly, in Section 4, we summarize the paper.

## 2. Database Schemas

### 2.1 Syntax

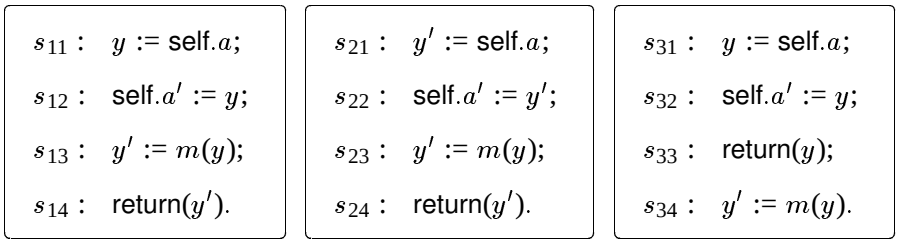
**Definition 1:** A *database schema* is a 6-tuple **S** = ($C, \leq, Attr, Ad, Meth, Impl$) where:

1. $C$ is a finite set of *class names*.
2. $\leq$ is a partial order on $C$ representing a *class hierarchy*. If $c' \leq c$, then we say that $c'$ is a *subclass* of $c$ and $c$ is a *superclass* of $c'$. For simplicity, we assume that the class hierarchy is a forest on $C$, that is, for all $c_1, c_2, c \in C$, either $c_1 \leq c_2$ or $c_2 \leq c_1$ whenever $c \leq c_1$ and $c \leq c_2$.
3. *Attr* is a finite set of *attribute names*.
4. $Ad : C \times Attr \rightarrow C$ is a partial function representing *attribute declarations*. By $Ad(c, a) = c'$, we mean that the value of attribute $a$ of an object of $c$ must be an object of $c'$ or its subclass.
5. *Meth* is a finite set of *method names*.
6. $Impl : C \times Meth \rightarrow WFP$ is a partial function representing *method implementations*, where *WFP* is the set of *well-formed programs* defined below.

A *sentence* is an expression which has one of the following forms:

1. $y := y'$,
2. $y :=$ self,
3. $y :=$ self.$a$,
4. $y := m(y')$,
5. self.$a := y'$,
6. return($y'$),

where $y$, $y'$ are *variables*, $a$ is an attribute name, $m$ is a method name, and self is a reserved word that denotes the object on which a method is invoked (or, to which a message is sent). A




(a) (b) (c)

Fig. 2 Example of Programs.

sentence of type 5 is called an *update operation*. The intuitive meaning of each sentence seems to be obvious and the formal semantics will be presented in Section 2.2.

A *program* is a finite sequence of sentences. We say that a program $s_1; s_2; \cdots; s_n$ is *well-formed* when the following two conditions hold:

- No undefined variable is referred to. That is, for each $s_i$ ($1 \leq i \leq n$), if $s_i$ is one of $y := y'$, $y := m(y')$, self.$a := y'$, and return($y'$), then there exists a sentence $s_j$ ($j < i$) that must be one of $y' := y''$, $y' :=$ self, $y' :=$ self.$a'$, and $y' := m'(y'')$, where $y''$ is a variable, $a'$ is an attribute name, and $m'$ is a method name.

- Only the last sentence $s_n$ must have the form return($y'$) for some variable $y'$. Thus the other sentences $s_1$, $s_2$,..., $s_{n-1}$ must be one of types 1 to 5. □

**Example 1:** Consider the three programs in Fig. 2. Program (a) is well-formed while (b) is not, since sentence $s_{23}$ refers to variable $y$ but no value is assigned to $y$ in any of the preceding sentences $s_{21}$ and $s_{22}$. Neither is program (c) since the last sentence $s_{34}$ is not in the form of return($y'$). □

We often omit temporary variables for readability. For example, we write "$y := m$(self.$a$)" instead of "$y'$ := self.$a$; $y := m(y')$," where $y'$ is a temporary variable.

**Definition 2:** The *description size* of **S**, denoted $\|\mathbf{S}\|$, is defined as follows:

$$\begin{aligned}\|\mathbf{S}\| = \quad & |C| + |\mathit{Attr}| + |\mathit{Meth}| \\ & + \text{(the number of attribute declarations given by } \mathit{Ad}) \\ & + \text{(the total number of sentences given by } \mathit{Impl}),\end{aligned}$$

where $|X|$ is the cardinality of a set $X$. □

### 2.2 Semantics

The *inherited implementation* of method $m$ at class $c$, denoted $\mathit{Impl}^*(c, m)$, is defined as $\mathit{Impl}(c', m)$ such that $c'$ is the smallest superclass of $c$ (with respect to the partial order $\leq$) at which an implementation of $m$ exists, that is, if $\mathit{Impl}(c'', m)$ is defined and $c \leq c''$, then it must hold that $c' \leq c''$. If such an implementation does not exist, then $\mathit{Impl}^*(c, m)$ is undefined. Similarly, the *inherited attribute declaration* of attribute $a$ at class $c$, denoted $\mathit{Ad}^*(c, a)$, is defined as $\mathit{Ad}(c', a)$ such that $c'$ is the smallest superclass of $c$ at which an attribute declaration of $a$ exists. If such an attribute declaration does not exist, then $\mathit{Ad}^*(c, a)$ is undefined.

A *database instance* of **S** is a pair $\mathbf{I} = (\nu, \mu)$, where:

1. To each class $c \in C$, $\nu$ assigns a disjoint, finite set $\nu(c)$ of *objects* (or *object identifiers*). Each $o \in \nu(c)$ is called an object of class $c$. Let $O_{\mathbf{S},\mathbf{I}} = \cup_{c \in C} \nu(c)$. Let $\mathit{cl}(o)$ denote the class $c$ such that $o \in \nu(c)$.

2. To each object $o \in \nu(c)$ and each attribute $a \in \mathit{Attr}$ such that $\mathit{Ad}^*(c, a)$ is defined, $\mu$ assigns an object, denoted $\mu(o, a)$, that is the *value* of attribute $a$ (or simply $a$-value) of $o$. If $\mathit{Ad}^*(c, a) = c'$, then $\mu(o, a)$ must belong to $\nu(c'')$ for some $c''$ ($c'' \leq c'$). Hereafter, $\mu(o, a)$ is often denoted by $o.a$.

The *operational semantics* of **S** is originally defined through a *method execution tree* [11]. In this paper, we present a more straightforward definition, in which the execution of a method is defined by rewriting rules on configurations of an interpreter for method implementations.

**Definition 3:** A *configuration* is one of the expressions

$$\langle \mu, o \rangle, \quad \mathit{active}(\mu, o, m, i, \sigma), \quad \mathit{CF} \circ \mathit{await}(o, m, i, \sigma),$$

where $\mu$ is an assignment representing attribute values, $o$ is an object, $m$ is a method name, $i$ is a positive integer, $\sigma$ is an assignment of objects to the variables appearing in *Impl*, and *CF* is a configuration. An *initial configuration* has the form *active*$(\mu, o, m, 1, \sigma_\perp)$, where *Impl*$^*$(*cl*$(o), m)$ is defined and $\sigma_\perp$ is an assignment undefined everywhere. □

Before presenting the formal semantics of configurations, we give an informal explanation here. *active*$(\mu, o, m, i, \sigma)$ means that the interpreter is about to execute the $i$-th sentence of *Impl*$^*$(*cl*$(o), m)$, where self in *Impl*$^*$(*cl*$(o), m)$ is interpreted as $o$, the current variable assignment is given by $\sigma$, and the current database instance is given by $\mu$. *CF* $\circ$ *await*$(o, m, i, \sigma)$ represents that another method has been invoked at the $i$-th sentence of *Impl*$^*$(*cl*$(o), m)$. $\langle \mu, o \rangle$ is the pair of the resulting database instance and the returned value after an execution of a method.

**Definition 4:** Let $s(c, m, i)$ denote the $i$-th sentence of *Impl*$^*(c, m)$. Let $f[(a_1, \ldots, a_n)/b]$ denote the function $f'$ that is equal to $f$ except that $f'(a_1, \ldots, a_n) = b$. The *one-step execution relation* $\rightarrow$ on configurations is defined by the rewriting rules shown in Fig. 3. Note that the execution is deterministic, that is, for every configuration *CF*, there is at most one *CF*$'$ such that *CF* $\rightarrow$ *CF*$'$. □

**Definition 5:** Let $o \in O_{\mathbf{S},\mathbf{I}}$. A *partial execution* of method $m$ for object $o$ under instance $\mathbf{I} = (\nu, \mu)$ is a (possibly infinite but non-empty) sequence $EX = \langle CF_0, CF_1, \ldots \rangle$ of configurations such that $CF_0$ is the initial configuration *active*$(\mu, o, m, 1, \sigma_\perp)$ and $CF_i \rightarrow CF_{i+1}$ for all $i$.

A partial execution *EX* is said to be *terminating* if $EX = \langle CF_0, \ldots, CF_n \rangle$ is a finite sequence and there is no $CF_{n+1}$ such that $CF_n \rightarrow CF_{n+1}$. If on the other hand *EX* is an infinite sequence, then *EX* is said to be *nonterminating*. Furthermore, *EX* is said to be *complete* if it is either terminating or nonterminating. □

**Definition 6:** A terminating execution $EX = \langle CF_0, \ldots, CF_n \rangle$ is *successful* if $CF_n = \langle \mu', o' \rangle$ for some $\mu'$ and $o'$, and *aborted* otherwise. □

Aborted executions are caused by two types of sentences "$y := m'(y')$" and "self$.a := y'$." By the rewriting rule (R4), an execution is aborted if method $m'$ is undefined for the class of the

(R1) If $s(cl(o), m, i)$ = "$y := y'$,"

$$active(\mu, o, m, i, \sigma) \rightarrow active(\mu, o, m, i + 1, \sigma[y/\sigma(y')]).$$

(R2) If $s(cl(o), m, i)$ = "$y :=$ self,"

$$active(\mu, o, m, i, \sigma) \rightarrow active(\mu, o, m, i + 1, \sigma[y/o]).$$

(R3) If $s(cl(o), m, i)$ = "$y :=$ self.$a$,"

$$active(\mu, o, m, i, \sigma) \rightarrow active(\mu, o, m, i + 1, \sigma[y/\mu(o, a)]).$$

(R4) If $s(cl(o), m, i)$ = "$y := m'(y')$" and $Impl^*(cl(\sigma(y')), m')$ is defined,

$$active(\mu, o, m, i, \sigma) \rightarrow active(\mu, \sigma(y'), m', 1, \sigma_\perp) \circ await(o, m, i, \sigma).$$

(R5) If $s(cl(o), m, i)$ = "self.$a := y'$" and $cl(\sigma(y')) \leq Ad^*(cl(o), a)$,

$$active(\mu, o, m, i, \sigma) \rightarrow active(\mu[(o, a)/\sigma(y')], o, m, i + 1, \sigma).$$

(R6) If $s(cl(o), m, i)$ = "return($y'$),"

$$active(\mu, o, m, i, \sigma) \rightarrow \langle \mu, \sigma(y') \rangle.$$

(R7) If $s(cl(o), m, i)$ = "$y := m'(y')$,"

$$\langle \mu, o' \rangle \circ await(o, m, i, \sigma) \rightarrow active(\mu, o, m, i + 1, \sigma[y/o']).$$

Fig. 3 One-Step Execution Relation.

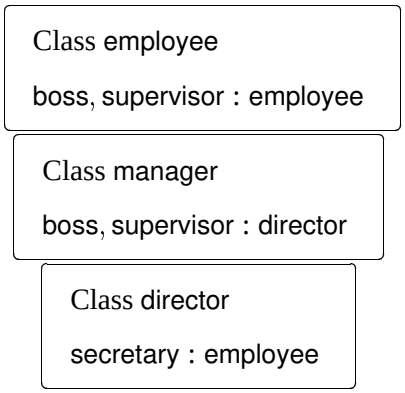



Fig. 4 Definition of $Ad_1$.

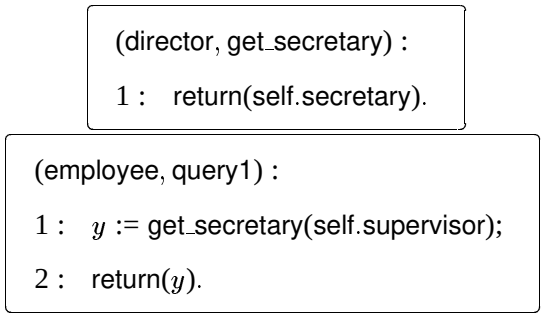



Fig. 5 Definition of $Impl_1$.

object assigned to $y'$. By (R5), an execution is aborted if the class of the object assigned to $y'$ violates the attribute declaration given by *Ad*. Both cases are viewed as *type errors*. Now we are ready to define the notions of consistency and termination.

**Definition 7:** **S** is *consistent* if every terminating execution is successful under every instance of **S**, and **S** is *terminating* if every complete execution is terminating under every instance of **S**. □

**Example 2:** Consider a database schema $\mathbf{S}_1 = (C_1, \leq_1, Attr_1, Ad_1, Meth_1, Impl_1)$, where

- $C_1 = \{\mathsf{director}, \mathsf{manager}, \mathsf{employee}\}$ and director $\leq_1$ manager $\leq_1$ employee;
- $Attr_1 = \{\mathsf{boss}, \mathsf{supervisor}, \mathsf{secretary}\}$ and $Ad_1$ is shown in Fig. 4; and
- $Meth_1 = \{\mathsf{get_secretary}, \mathsf{query1}\}$ and $Impl_1$ is shown in Fig. 5.

Fig. 6 illustrates a database instance $\mathbf{I}_1 = (\nu_1, \mu_1)$ of $\mathbf{S}_1$, where Bob, Sara,... are objects and Bob $\rightarrow$ Sara means $\mu_1(\mathsf{Bob}, \mathsf{boss}) = \mu_1(\mathsf{Bob}, \mathsf{supervisor}) = \mathsf{Sara}$. Consider the execution of query1 for Bob. Since $\mu_1(\mathsf{Bob}, \mathsf{supervisor}) = \mathsf{Sara} \in \nu_1(\mathsf{manager})$ and $Impl_1^*(\mathsf{manager}, \mathsf{get_secretary})$ is undefined, the execution is aborted. Also it is easily checked that $\mathbf{S}_1$ is terminating.

Let $\mathbf{S}_1' = (C_1, \leq_1, Attr_1, Ad_1, Meth_1', Impl_1')$, where $Meth_1' = \{\mathsf{calc_supervisor}, \mathsf{get_secretary}, \mathsf{query2}\}$ and $Impl_1'$ is shown in Fig. 7. $\mathbf{I}_1$ is also an instance of $\mathbf{S}_1'$. The execution of

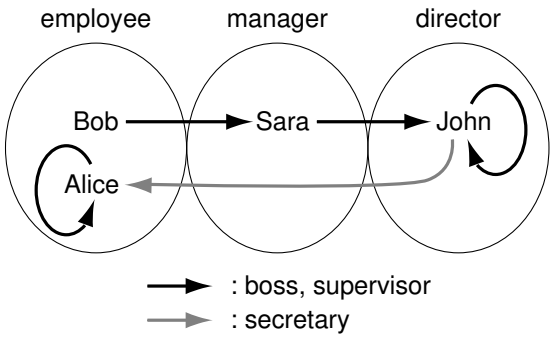



Fig. 6 A Database Instance $\mathbf{I}_1$.

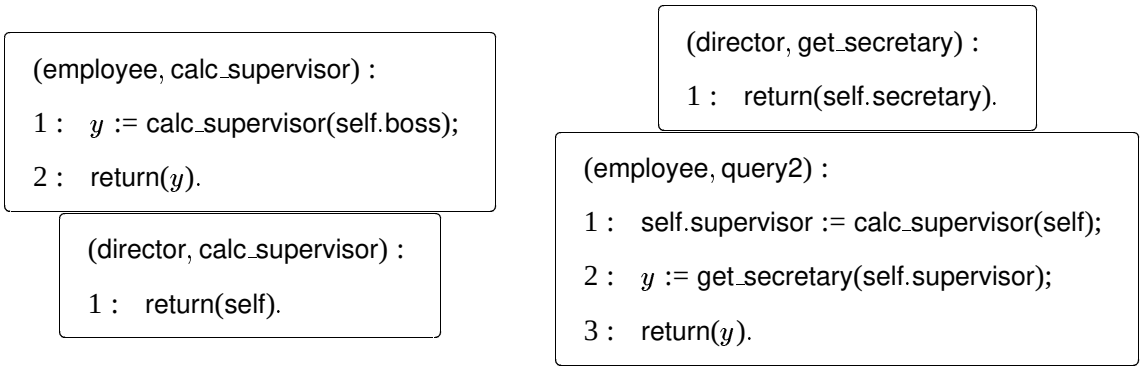



Fig. 7 Definition of $Impl'_1$.

calc_supervisor for Bob is successful and the last configuration is $\langle \mu_1, \mathsf{John} \rangle$, i.e., the returned value of the execution is John. On the other hand, the execution of calc_supervisor for Alice is nonterminating. It can be shown that calc_supervisor returns an object of class director when it terminates. Next, consider the execution of query2 for Bob. When control reaches the second sentence of (employee, query2) in Fig. 7, Bob.supervisor has been set to John $\in \nu_1$(director). Therefore the execution is successful. Consequently, it can be proved that $\mathbf{S}'_1$ is consistent. □

**Example 3:** Consider a database schema $\mathbf{S}_2 = (C_2, \leq_2, Attr_2, Ad_2, Meth_2, Impl_2)$, where

- $C_2 = \{c, c_\mathrm{t}, c_\mathrm{f}\}$ such that $c_\mathrm{t} \leq_2 c$ and $c_\mathrm{f} \leq_2 c$ (i.e., $c$ is a superclass of both $c_\mathrm{t}$ and $c_\mathrm{f}$, see Fig. 8(a)); and

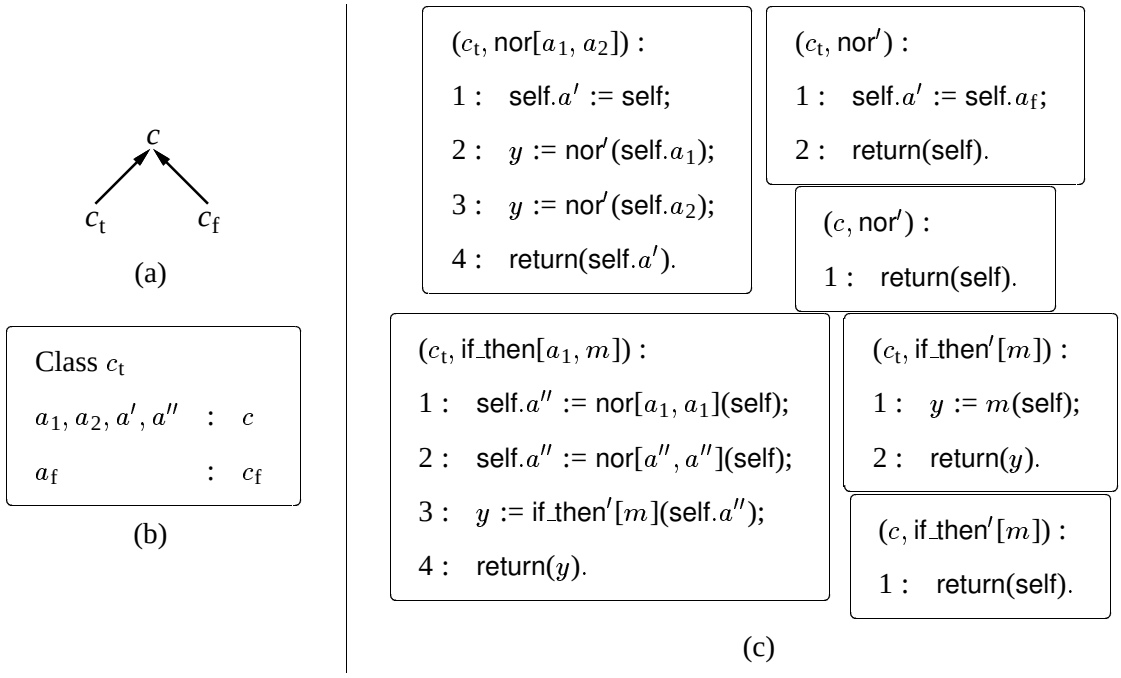



Fig. 8 Definition of $\mathbf{S}_2$.

- $Ad_2$ is shown in Fig. 8(b).

We adopt the following Boolean-value representation: Let $o$ be an object of class $c_\mathrm{t}$. Each attribute $a \in \{a_1, a_2, a', a'', a_\mathrm{f}\}$ of $o$ represents true if $o.a = o$, and false otherwise. Note that $o.a_\mathrm{f}$ always represents false because of the declaration $Ad_2(c_\mathrm{t}, a_\mathrm{f}) = c_\mathrm{f}$.

Then, we define two methods nor$[a_1, a_2]$ and if_then$[a_1, m]$ as shown in Fig. 8(c). Method nor$[a_1, a_2]$ calculates NOR of $o.a_1$ and $o.a_2$, and returns $o$ if the result is true and $o.a_\mathrm{f}$ otherwise. Since every Boolean operator can be represented by NORs, we can construct a method which calculates a given Boolean formula using nor$[a_1, a_2]$. On the other hand, if_then$[a_1, m]$ simulates if-then statements: $m$ is invoked on $o$ if and only if $o.a_1 = o$. By the first two lines of $(c_\mathrm{t}, \mathsf{if_then}[a_1, m])$, $o.a''$ is "normalized" so that $o.a'' = o.a_\mathrm{f}$ (and hence $cl(o.a'') \neq c_\mathrm{t}$) whenever $o.a_1$ represents false. □

### 2.3 Subclasses of the Database Schemas

In the last part of this section, we provide some notions to define subclasses of the database schemas.

**Definition 8:** The *height* of $\leq$ is the maximum integer $n$ such that there exist distinct $c_0$, $c_1$,...,$c_n \in C$ satisfying $c_0 \leq c_1 \leq \cdots \leq c_n$. □

If the height of $\leq$ is zero, then the class hierarchy is *flat*. That is, all classes are completely separated and there is no superclass-subclass relation at all. We often say that **S** is *flat* if $\leq$ is flat.

**Definition 9:** *Ad* is *covariant* if $c_1 \leq c_2$ implies *Ad**$(c_1, a) \leq$ *Ad**$(c_2, a)$ for all $c_1, c_2 \in C$ and $a \in$ *Attr* such that both *Ad**$(c_1, a)$ and *Ad**$(c_2, a)$ are defined. □

Usually, covariance is defined as a property of method signatures. For example, in [2], a schema is said to be covariant if for each built-in method $m$ (assumed to be monadic for simplicity) and for each pair $(m : c_1 \rightarrow c_1')$, $(m : c_2 \rightarrow c_2')$ of signatures of $m$, we have that $c_1' \leq c_2'$ whenever $c_1 \leq c_2$. In our model, an attribute $a$ can be regarded as a built-in method $m_a$ such that the signatures of $m_a$ are given by *Ad* and the interpretation of $m_a$ is given by a database instance.

There are many situations in which it is natural to assume the covariance. For example, technical_paper $\leq$ literature and *Ad**(technical_paper, author) $\leq$ *Ad**(literature, author), the latter of which means that the authors of technical-papers are a subclass of those of general literatures.

**Definition 10:** *Impl* is *retrieval* if it includes no update operation (i.e., sentence in the form of "self.$a$ := $y$"). We often say that **S** is *retrieval* if *Impl* is retrieval. □

**Definition 11:** The *method dependency graph* $G = (V, E)$ of *Impl* is defined as follows [2]:

- $V$ = *Meth*; and
- An edge from $m$ to $m'$ is in $E$ if and only if there is a class $c$ such that $m$ appears in *Impl*$(c, m')$.

If the method dependency graph of *Impl* is acyclic, then *Impl* is *recursion-free*. We often say that **S** is *recursion-free* if *Impl* is recursion-free. Note that **S** is terminating whenever it is recursion-free. □

## 3. Complexity of the Type-Consistency Problem

### 3.1 Non-Flatness of the Class Hierarchy

In this section, we show how non-flatness of the class hierarchy affects the complexity of the type-consistency problem. First, the following theorem claims that consistency for a flat schema is solvable in polynomial time.

**Theorem 1:** Let $\mathbf{S} = (C, \leq, \mathit{Attr}, \mathit{Ad}, \mathit{Meth}, \mathit{Impl})$ be a database schema. If **S** is flat, then consistency for **S** is solvable in polynomial time.

*Proof*: Define an instance $\tilde{\mathbf{I}} = (\tilde{\nu}, \tilde{\mu})$ of **S** as follows:

- $\tilde{\nu}(c) = \{o_c\}$ for each $c \in C$; and
- $\tilde{\mu}(o_c, a) = o_{c'}$ if $\mathit{Ad}^*(c, a) = c'$.

Note that $\tilde{\mu}$ is never altered during any execution even if **S** is not retrieval, since $\leq$ is flat and each class has exactly one object.

First, we show that there is an aborted execution under $\tilde{\mathbf{I}}$ if and only if **S** is inconsistent. The "only if" part is obvious. Conversely, let **I** $(= (\nu, \mu))$ be an arbitrary instance of **S** and $h : O_{\mathbf{S},\mathbf{I}} \rightarrow O_{\mathbf{S},\tilde{\mathbf{I}}}$ be a homomorphism such that $h(o) = o_c$ for each $o \in \nu(c)$. It can be shown that for every (partial) execution *EX* under **I**, $h(EX)$ is a (partial) execution under $\tilde{\mathbf{I}}$ by induction on the length of *EX*. Then, it can be easily proved that $h(EX)$ is aborted whenever *EX* is aborted.

To check whether there is an aborted execution under $\tilde{\mathbf{I}}$, compute the last configuration of the execution of each $m$ for each $o \in O_{\mathbf{S},\tilde{\mathbf{I}}}$, not the entire execution, since computing the entire executions takes exponential time in general. We use a table $T$, where $T(o_c, m, i)$ represents the last configuration of the partial execution from the first sentence up to the $i$-th sentence in $\mathit{Impl}^*(c, m)$. Define $T(o_c, m, 0)$ as $\mathit{active}(\tilde{\mu}, o_c, m, 1, \sigma_\perp)$. If $s(c, m, i)$ is not $y := m'(y')$, compute $T(o_c, m, i+1)$ from $T(o_c, m, i)$ through the corresponding rewriting rule in Fig. 3. Suppose

that $s(c, m, i+1)$ = "$y := m'(y')$." Also suppose that $T(o_c, m, i)$ = *active*$(\tilde{\mu}, o_c, m, i, \sigma)$ for some $\sigma$. If *Impl*$^*(cl(\sigma(y')), m')$ is undefined, then the execution of $m$ for $o_c$ is aborted. Otherwise, there are the following three cases. Let $n$ be the number of sentences in *Impl*$^*(cl(\sigma(y')), m')$.

1. If we have already obtained $T(\sigma(y'), m', n)$, then compute $T(o_c, m, i+1)$ through (R7).
2. Suppose that $T(\sigma(y'), m', n)$ has not been obtained yet.
   (a) If we have already tried to compute $T(\sigma(y'), m', 1)$, then give up computing $T(\sigma(y'), m', n)$ (and thus $T(o_c, m, i+1)$), since this execution is nonterminating.
   (b) Otherwise, try to compute $T(\sigma(y'), m', 1), \ldots, T(\sigma(y'), m', n)$.

In summary, for all $c$ and $m$ such that *Impl*$^*(c, m)$ is defined, compute $T(o_c, m, i)$ in a depth-first manner. Since each $T(o_c, m, i)$ is computed at most once, this algorithm terminates in a linear time of the size of $T$. And $T$ has a linear size of the total number of sentences given by *Impl* since flatness implies *Impl* = *Impl*$^*$. □

On the other hand, the following theorem says that consistency for a non-flat schema is undecidable even if it is retrieval and the height of the class hierarchy is bounded by one.

**Theorem 2:** Let **S** = $(C, \leq, Attr, Ad, Meth, Impl)$ be a non-flat database schema. Consistency for **S** is undecidable, even if **S** is retrieval, the height of $\leq$ is one, and *Ad* is covariant. □

This theorem is proved by showing a reduction from the Post's Correspondence Problem (PCP) to the consistency problem for a database schema which satisfies the assumption in the theorem. Let $\langle w, u \rangle$ ($w = \langle w_1, \ldots, w_n \rangle$, $u = \langle u_1, \ldots, u_n \rangle$) be an instance of PCP over alphabet $\Sigma = \{0, 1\}$. We construct a database schema $\mathbf{S}_{w,u}$ such that

- $\mathbf{S}_{w,u}$ is retrieval;
- the height of $\leq$ of $\mathbf{S}_{w,u}$ is one;
- *Ad* of $\mathbf{S}_{w,u}$ is covariant; and
- $\mathbf{S}_{w,u}$ is inconsistent if and only if $\langle w, u \rangle$ has a solution.

The idea for $\mathbf{S}_{w,u}$ to satisfy the last condition is as follows. Let post be a method in $\mathbf{S}_{w,u}$, which plays the principal role in the reduction. Each pair of a database instance $\mathbf{I}$ and an object $o_1 \in O_{\mathbf{S},\mathbf{I}}$ is regarded as a candidate for a solution of $\langle w, u \rangle$. If $(\mathbf{I}, o_1)$ is actually a solution of $\langle w, u \rangle$, then the execution of post for $o_1$ under $\mathbf{I}$ is aborted. Otherwise, the execution of post for $o_1$ under $\mathbf{I}$ is nonterminating (therefore no type error occurs during the execution). By ensuring that no type error occurs during the execution of any method except post, we can conclude that $\mathbf{S}_{w,u}$ satisfies the last condition.

Now we show the construction of $\mathbf{S}_{w,u}$. Suppose that

$$\begin{array}{lllllll} w_1 & = & w_{1,1}w_{1,2}\cdots w_{1,d_1}, & \ldots, & w_n & = & w_{n,1}w_{n,2}\cdots w_{n,d_n}, \\ u_1 & = & u_{1,1}u_{1,2}\cdots u_{1,e_1}, & \ldots, & u_n & = & u_{n,1}u_{n,2}\cdots u_{n,e_n}, \end{array}$$

where all of the $w_{i,j}$'s and $u_{i,j}$'s are in $\Sigma$. Figs. 9 and 10 show the definition of $\leq$ and *Ad* of $\mathbf{S}_{w,u}$, respectively. Class $c_i$ $(1 \leq i \leq n)$ represents the $i$-th pair $\langle w_i, u_i \rangle$, and class $c'_0$ (resp. $c'_1$) represents symbol 0 (resp. 1). Note that the height of $\leq$ is one and *Ad* is covariant. Next, define methods post, $m_w$, is0, is1, and isc′ as Figs. 11–14 (also define method $m_u$ similarly to $m_w$). The underlined part (e.g., the second line of $(c_i, m_w)$) is a macro notation, and all of them can be expanded when $\langle w, u \rangle$ is reduced to $\mathbf{S}_{w,u}$. Note that $\mathbf{S}_{w,u}$ is retrieval (i.e., there is no sentence in the form of self.$a$ := $y$). Moreover,

- each method except post and test has its definition at every class;
- method post is not invoked by another method; and
- method test, which appears at the fifth line of $(c_i, \mathsf{post})$, has no definition at any class, and can be invoked only by post.

Thus, a type error occurs if and only if the control reaches the fifth line of $(c_i, \mathsf{post})$ during the execution of post. Therefore, in order to prove the correctness of the reduction, it suffices to show that $\langle w, u \rangle$ has a solution if and only if there is an instance $\mathbf{I}$ such that the control reaches the fifth line of $(c_i, \mathsf{post})$ during the execution of post for some $o_1 \in O_{\mathbf{S},\mathbf{I}}$ under $\mathbf{I}$.

Let $\mathbf{I} = (\nu, \mu)$ and $o_1 \in \nu(c_1) \cup \cdots \cup \nu(c_n)$. In what follows, we explain the behavior of the execution of post for $o_1$ under $\mathbf{I}$. First, assume that $\mathbf{I}$ is in the following form (F1) (see also Fig. 15):

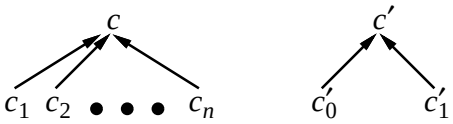



Fig. 9 $\leq$ of $\mathbf{S}_{w,u}$.

| Class $c_i$ $(1 \leq i \leq n)$ | Class $c$ | Class $c'_0$, $c'_1$ |
|---|---|---|
| $a_{\rightarrow}$ : $c$ | $a_{\Rightarrow}$ : $c'$ | $a_{\Rightarrow}$ : $c'$ |

Fig. 10 *Ad* of $\mathbf{S}_{w,u}$.

```
(c_i, post) (1 ≤ i ≤ n) :
1 :  y := m_w(self);
2 :  y := isc′(y);
3 :  y := m_u(self);
4 :  y := isc′(y);
5 :  y := test(self);
6 :  return(self).
```

Fig. 11 Definition of Method post.

(F1)
- $o_i.a_{\rightarrow} = o_{i+1} \in \nu(c_1) \cup \cdots \cup \nu(c_n)$ $(1 \leq i \leq k-1)$,
- $o_k.a_{\rightarrow} = o_{k+1} \in \nu(c)$,
- $o_{k+1}.a_{\Rightarrow} = o'_1 \in \nu(c'_0) \cup \nu(c'_1)$ and $o'_j.a_{\Rightarrow} = o'_{j+1} \in \nu(c'_0) \cup \nu(c'_1)$ $(1 \leq j \leq l-1)$,
- $o_l.a_{\Rightarrow} = o'_{l+1} \in \nu(c')$.

In **I**, sequence $o_1 \cdots o_k$ represents a candidate for a solution of $\langle w, u \rangle$, and sequence $o'_l \cdots o'_1$ represents a word over $\Sigma$. Let $w_{o_i}$ and $u_{o_i}$ denote the words represented by $o_i$ (i.e., $w_{o_i} = w_{i'}$ and $u_{o_i} = u_{i'}$ if $o_i \in \nu(c_{i'})$), and $x_j$ denote the symbol represented by $o'_j$ (i.e., $x_j = 0$ if $o'_j \in \nu(c'_0)$, and $x_j = 1$ if $o'_j \in \nu(c'_1)$). The following two lemmas claim that the execution of the first two

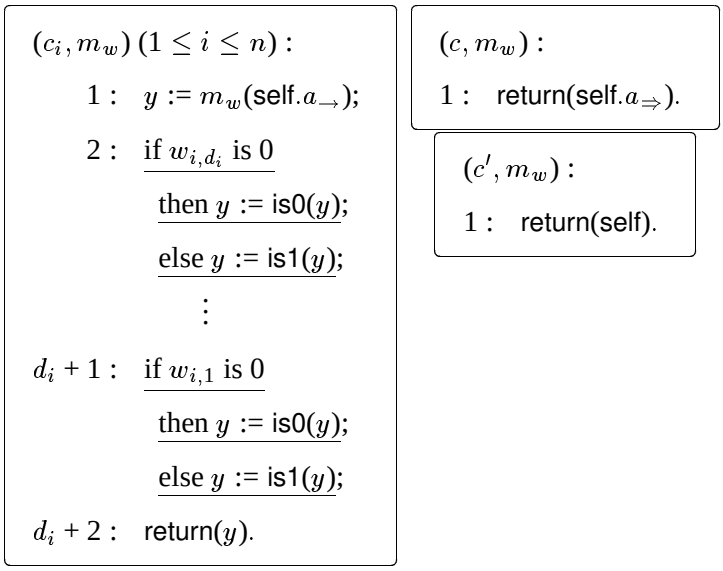



Fig. 12 Definition of Method $m_w$.

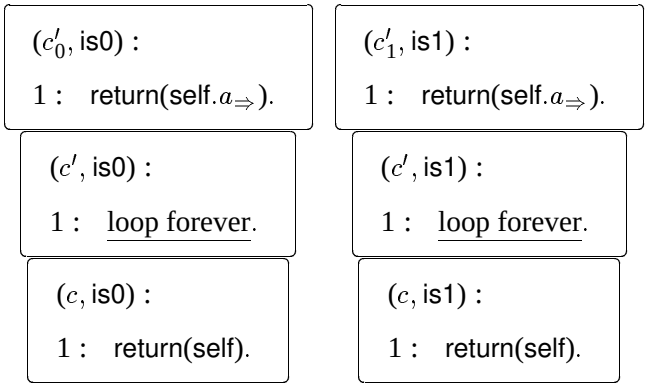



Fig. 13 Definition of Methods is0 and is1.

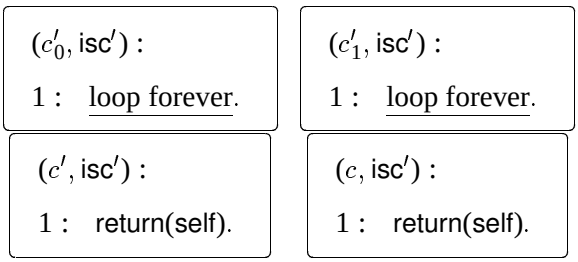



Fig. 14 Definition of Method isc$'$.

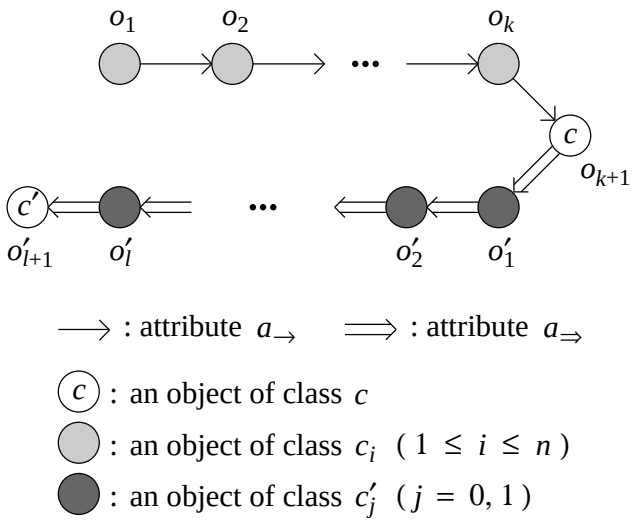



Fig. 15 A Database Instance of $\mathbf{S}_{w,u}$.

lines of ($cl(o_1)$, post) terminates if and only if $w_{o_1} \cdots w_{o_k} = x_l \cdots x_1$.

**Lemma 1:** Suppose that **I** is in the form of (F1). If there is $l'$ ($l' \leq l$) such that $w_{o_1} \cdots w_{o_k} = x_{l'} \cdots x_1$, then the execution of $m_w$ for $o_1$ terminates and returns $o'_{l'+1}$. Otherwise, the execution of $m_w$ for $o_1$ does not terminate.

*Proof*: The lemma is proved by induction on $k$. Without loss of generality, $o_1$ is assumed to be an object of class $c_1$.

[Basis] Suppose that $k = 1$. By the first line of $(c_1, m_w)$, method $m_w$ is recursively invoked on $o_1.a_\rightarrow$, which is an object of class $c$ since $k = 1$. By $(c, m_w)$, this invocation results in $o'_1$, and it is assigned to $y$ at the first line of $(c_1, m_w)$. Suppose that $w_{1,d_1} = 0$. By the second line of $(c_1, m_w)$, method is0 is invoked on $o'_1$. From the definition of is0, the execution of is0 for $o'_1$ terminates and returns $o'_1.a_\Rightarrow$ ($= o'_2$) if $o'_1 \in \nu(c'_0)$, and does not terminate if $o'_1 \in \nu(c') \cup \nu(c'_1)$. Since a similar property holds when $w_{1,d_1} = 1$, we can conclude that the execution of the second line of $(c_1, m_w)$ terminates and $o'_2$ is assigned to $y$ if and only if $w_{1,d_1} = x_1$. And by induction on $d_1$, we obtain that the execution of $m_w$ for $o_1$ terminates and returns $o'_{d_1+1}$ if $w_{o_1} = x_{d_1} \cdots x_1$ and $d_1 \leq l$, and does not terminate otherwise.

[Inductive Step] Suppose that $k > 1$. By the first line of $(c_1, m_w)$, method $m_w$ is recursively invoked on $o_1.a_{\rightarrow}$ (= $o_2$). From the inductive hypothesis, the execution of $m_w$ for $o_2$ terminates and returns $o'_{l''+1}$ if $w_{o_2} \cdots w_{o_k} = x_{l''} \cdots x_1$ and $l'' \leq l$, and does not terminate otherwise. In and after the second line of $(c_1, m_w)$, it is checked that $w_{o_1} = x_{l''+d_1} \cdots x_{l''+1}$ and $l'' + d_1 \leq l$. Thus, the lemma holds when $k > 1$. □

**Lemma 2:** Suppose that **I** is in the form of (F1). The execution of isc′ for $o'_{l'+1}$ terminates if and only if $o'_{l'+1} = o'_{l+1}$ (i.e., $l' = l$).

*Proof*: Obvious from the definition of isc′. □

Thus, the third line of $(cl(o_1), \mathsf{post})$ is executed if and only if $w_{o_1} \cdots w_{o_k} = x_l \cdots x_1$. And the similar lemmas hold for the third and fourth lines of $(cl(o_1), \mathsf{post})$. Therefore, the control reaches the fifth line of $(cl(o_1), \mathsf{post})$ if and only if $w_{o_1} \cdots w_{o_k} = u_{o_1} \cdots u_{o_k} = x_l \cdots x_1$.

Next, suppose that **I** is not in the form of (F1). Then, **I** must be in one of the forms (F2) and (F3):

(F2) The "$a_{\rightarrow}$-chain" forms a cycle. That is, there is $o \in \nu(c_1) \cup \cdots \cup \nu(c_n)$ such that $o_1.a_{\rightarrow} \ldots a_{\rightarrow} = o$ and $o.a_{\rightarrow} \ldots a_{\rightarrow} = o$.

(F3) The "$a_{\rightarrow}$-chain" does not form a cycle but the "$a_{\Rightarrow}$-chain" forms a cycle. That is, there are $o \in \nu(c)$ and $o' \in \nu(c'_0) \cup \nu(c'_1)$ such that $o_1.a_{\rightarrow} \ldots a_{\rightarrow} = o$, $o.a_{\Rightarrow} \ldots a_{\Rightarrow} = o'$, and $o'.a_{\Rightarrow} \ldots a_{\Rightarrow} = o'$.

In the case of (F2), the recursive call of $m_w$ at the first line of $(c_i, m_w)$ does not terminate. In the case of (F3), the execution of is0 or is1 in $(c_i, m_w)$, or isc′ in $(cl(o_1), \mathsf{post})$ does not terminate. Therefore, if **I** is not in the form of (F1), then the control does not reach the fifth line of $(cl(o_1), \mathsf{post})$.

Suppose that $\langle w, u \rangle$ has a solution. Then, there is an instance **I** in the form of (F1) such that $w_{o_1} \cdots w_{o_k} = u_{o_1} \cdots u_{o_k} = x_l \cdots x_1$. During the execution of post for $o_1$ under **I**, the control reaches the fifth line of $(cl(o_1), \mathsf{post})$. Conversely, suppose that there is an instance **I** such that the control reaches the fifth line of $(cl(o_1), \mathsf{post})$ during the execution of post for $o_1$ under **I**. Then,

**I** must be in the form of (F1) and satisfy that $w_{o_1} \cdots w_{o_k} = u_{o_1} \cdots u_{o_k} = x_l \cdots x_1$. Obviously, $o_1, \ldots, o_k$ represent the solution of $\langle w, u \rangle$. This concludes the proof of Thoerem 2.

As stated in Section 1, method schemas [2], [3] are based on a functional OOPL model. Since $\mathbf{S}_{w,u}$ is retrieval, it can be translated into a method schema. For example,

$$\begin{aligned} \mathit{Impl}(c_i, \mathsf{post}) &= \mathsf{test}(\mathsf{isc}'(m_w(\mathsf{self})), \mathsf{isc}'(m_u(\mathsf{self}))), \\ \mathit{Impl}(c_i, m_w) &= \mathsf{isX}_{i,1}(\cdots \mathsf{isX}_{i,d_i}(m_w(m_{a_\rightarrow}(\mathsf{self}))) \cdots), \end{aligned}$$

where $\mathsf{isX}_{i,j}$ is either $\mathsf{is0}$ or $\mathsf{is1}$ according to $w_{i,j}$, and $m_{a_\rightarrow}$ is a method which returns the $a_\rightarrow$-value of the argument object. It is easily verified that $\mathbf{S}_{w,u}$ can be translated into a method schema with methods of arity two. Thus, we have the following result which was open in [2]:

**Corollary 1:** Consistency for a method schema with methods of arity two is undecidable. □

### 3.2 Recursion

Intuitively, recursion makes the length of the execution unbounded. In this section, we show that the complexity of the type-consistency problem is affected by this unboundedness.

**Theorem 3:** Let **S** = ($C, \leq, \mathit{Attr}, \mathit{Ad}, \mathit{Meth}, \mathit{Impl}$) be a database schema with recursion. Consistency for **S** is undecidable, even if **S** is terminating, the height of $\leq$ is one, and *Ad* is covariant. □

To prove Theorem 3, for a given input string $x$ of a fixed deterministic Turing machine $M$, we construct a schema $\mathbf{S}_{M,x}$ satisfying the following conditions:

- $\mathbf{S}_{M,x}$ is terminating;
- the height of $\leq$ of $\mathbf{S}_{M,x}$ is one;
- *Ad* of $\mathbf{S}_{M,x}$ is covariant; and
- $\mathbf{S}_{M,x}$ is inconsistent if and only if $M$ accepts $x$.

First of all, we define a Turing machine and an instantaneous description.

**Definition 12:** A *deterministic Turing machine* $M$ is a triple $(Q, \Sigma, \delta)$, where

- $Q$ is a finite set of states. $Q$ contains two special states: the initial state $q_0$ and the accepting state $q_{\text{yes}}$;
- $\Sigma$ is a finite set of symbols. $\Sigma$ contains two special symbols: the *blank symbol* $B$ and the *first symbol* $\triangleright$. The first symbol is always placed at the leftmost cell of the tape;
- $\delta$ is a function which maps $(Q - \{q_{\text{yes}}\}) \times \Sigma$ to $Q \times \Sigma \times \{\leftarrow, \rightarrow, -\}$. We assume that if $\delta(q, \triangleright) = (q', \gamma, d)$, then $\gamma = \triangleright$ and $d = \rightarrow$. Therefore, the tape head never falls off the left end of the tape.

An *instantaneous description* (ID) $I$ of $M$ is a finite sequence $\langle q_1, \gamma_1 \rangle, \ldots, \langle q_k, \gamma_k \rangle$, where $q_i \in Q \cup \{\bot\}$ and $\gamma_i \in \Sigma$. It is required that $\gamma_1 = \triangleright$, and exactly one $q_i$ is in $Q$ ($i$ denotes the head position). The $i$-th pair $\langle q_i, \gamma_i \rangle$ of an ID $I$ is denoted by $I[i]$. The transition relation $\vdash_M$ over the set of IDs is defined as usual. □

We only describe the outline of the reduction (see Appendix A for a complete proof). First, in order to ensure that the execution of each recursively-defined method $m$ is terminating, we use an attribute, say $a_{\text{ws}}$, which "marks" an object. Suppose that an object $o$ is visited by a recursive invocation of $m$. If $o.a_{\text{ws}}$ represents true (see Example 3), then $m$ sets $o.a_{\text{ws}}$ false and continue the execution. Otherwise, $m$ returns from the invocation. Consequently, $o.a_{\text{ws}}$ represents true only if $o$ has not been visited. Since the set $O_{\mathbf{S}_{M,x},\mathbf{I}}$ of objects is finite, it can be shown that $\mathbf{S}_{M,x}$ is terminating. Moreover, by setting $o.a_{\text{ws}}$ true when $m$ returns, other recursively-defined methods can reuse $a_{\text{ws}}$. See Lemma 3 in Appendix A for a formal description of this technique.

Let TM be a method in $\mathbf{S}_{M,x}$, which plays the principal role in the reduction. TM simulates $M$ on $x$ as follows. Each database instance $\mathbf{I}$ of $\mathbf{S}_{M,x}$ is considered as a working space to compute the IDs of $M$ on $x$. TM simulates $M$ on $x$ exactly $r$ steps, where $r \geq 0$ is a constant dependent on $\mathbf{I}$. If the ID after $r$-step transitions contains the accepting state $q_{\text{yes}}$, then TM causes a type error. Otherwise, the execution of TM is successful. By ensuring that no type error occurs during the execution of any method except TM, the following property holds: If $M$ accepts $x$, then there is an instance $\mathbf{I}$ such that both the number of steps $r$ and the size of the working space determined

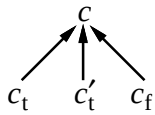



Fig. 16 $\leq$ of $\mathbf{S}_{M,x}$.

by **I** are large enough to find an aborted execution of TM under **I** (i.e., $\mathbf{S}_{M,x}$ is inconsistent). Otherwise, there is no aborted execution of TM under any instance (i.e., $\mathbf{S}_{M,x}$ is consistent).

Define $\leq$ and *Ad* of $\mathbf{S}_{M,x}$ as shown in Figs. 16 and 17, respectively. In Fig. 17, $\vec{a}$ denotes a tuple $(a_1, \ldots, a_K)$ of attributes, where $K = \lceil \log((|Q| + 1)|\Sigma|) \rceil$ (i.e., the number of bits to represent an element of an ID). $Ad(c_{\mathrm{t}}, \vec{a}) = c$ means that $Ad(c_{\mathrm{t}}, a_i) = c$ for each $i$ $(1 \leq i \leq K)$. An element of an ID is stored in $\vec{a}$ as the binary encoded form stated in Example 3. Attributes $\vec{a}'$ and $\vec{a}''$ are used for storing intermediate results during the computation of an ID. Attribute $a_{\mathrm{cont}}$ is used for determining $r$, i.e., the number of steps to be simulated. Attributes $a_{\mathrm{yes}}$ and $a'_{\mathrm{yes}}$ are used for checking whether $M$ is in the accepting state or not after the simulation. Note that the height of $\leq$ is one and *Ad* is covariant. Next, define method TM as shown in Fig. 18. All the methods except test is defined at every class. Method test is defined only at class $c_{\mathrm{f}}$. Since we can define all the methods so that no update operation causes a type error (see the method definitions presented in Appendix A), a type error occurs if and only if the control reaches the fifth line of ($c_{\mathrm{t}}$, TM) and test is about to be invoked on an object of class $c$, $c_{\mathrm{t}}$, or $c'_{\mathrm{t}}$.

In what follows, we explain the behavior of TM. Let $\mathbf{I} = (\nu, \mu)$ be a database instance of $\mathbf{S}_{M,x}$ and $o_1 \in \nu(c_{\mathrm{t}})$. Suppose that TM is invoked on $o_1$. Then get_ws is executed for $o_1$ by the first line of ($c_{\mathrm{t}}$, TM). This obtains objects $o_2, \ldots, o_{k+1}$ satisfying $o_i.a_{\Rightarrow} = o_{i+1}$ $(1 \leq i \leq k)$ by following attribute $a_{\Rightarrow}$ of each $o_i$, where $k$ is a constant dependent on **I** and satisfies $k \geq 1$. The objects $o_2, \ldots, o_{k+1}$ will be used as a working space to simulate $M$. Since attribute $a_{\Rightarrow}$ is defined only at class $c_{\mathrm{t}}$, the class of $o_2, \ldots, o_k$ must be $c_{\mathrm{t}}$. By a technical reason, we want $o_{k+1}$ to be an object of class $c'_{\mathrm{t}}$. To achieve this, if the $a_{\Rightarrow}$-chain from $o_1$ (1) ends up with an object of class $c_{\mathrm{f}}$ or $c$, or (2) forms a cycle, then get_ws changes the value of $o_k.a_{\Rightarrow}$ to an object of class $c'_{\mathrm{t}}$ (see Fig. 19). Lemma 5 in Appendix A provides a formal description of the behavior of get_ws.

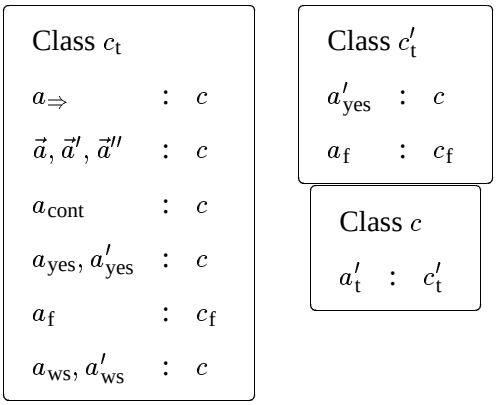



Fig. 17 *Ad* of $\mathbf{S}_{M,x}$.

```
(c_t, TM) :
1 :  y := get_ws(self);
2 :  y := initws(self);
3 :  y := step(self);
4 :  y := accept(self);
5 :  y := test(y);
6 :  return(self).
```

Fig. 18 Definition of Method TM.

Let $I_0$ be the initial ID of $M$ on $x$, and $n$ be the length of $I_0$. By executing initws for $o_1$ at the second line of ($c_{\mathrm{t}}$, TM), each $I_0[i]$ ($1 \leq i \leq k$) is stored in $o_i.\vec{a}$, where $o_i.\vec{a}$ denotes the tuple $(o_i.a_1, \ldots, o_i.a_K)$. Therefore, if $k < n$, then elements $I_0[k+1], \ldots, I_0[n]$ are abandoned. Conversely, if $n < k$, then $\langle \perp, B \rangle$ is stored in $o_{n+1}.\vec{a}, \ldots, o_k.\vec{a}$ (Actually, this is done by get_ws; see the definitions of get_ws and initws presented in Appendix A). Lemma 6 in Appendix A provides a formal description of the behavior of initws.

Method step simulates $r$-step transitions of $M$. Let $I_j$ denote the $j$-th ID of $M$ on $x$ (counting from zero). Suppose that the first $k - j$ elements of $I_j$ are stored in $o_{j+1}.\vec{a}, \ldots, o_k.\vec{a}$. More precisely, $I_j[i]$ ($1 \leq i \leq k - j$) is stored in $o_{j+i}.\vec{a}$. Note that the initial ID $I_0$ satisfies this

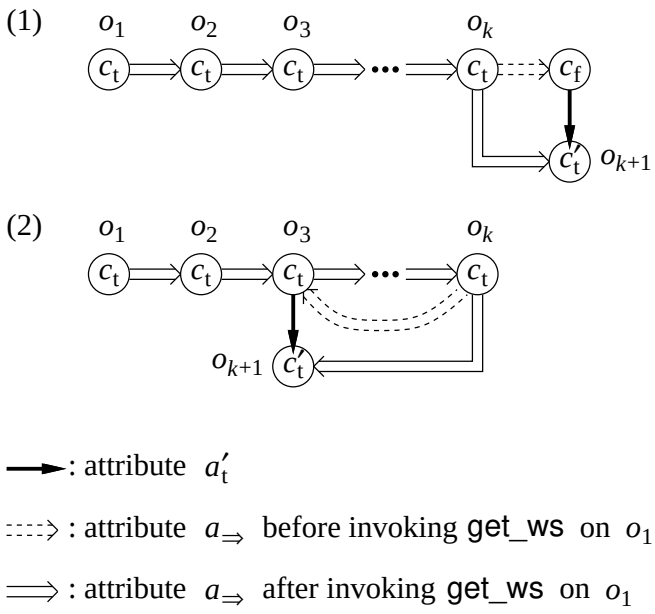



Fig. 19 A Database Instance after Invoking Method get_ws on $o_1$.

condition. Consider a database instance shown in Fig. 20(a). Let us compute the next ID $I_{j+1}$. Note that $I_{j+1}[i]$ can be computed from $I_j[i-1]$, $I_j[i]$, and $I_j[i+1]$. Therefore, if these three adjacent elements are stored in one object, we can compute $I_{j+1}[i]$ using nor$[*,*]$ stated in Example 3. To do this, for every object $o$ in the $a_\Rightarrow$-chain, we copy the element of the ID stored in $o$ to $o.a_\Rightarrow$ and $o.a_\Rightarrow.a_\Rightarrow$ as shown in Fig. 20(b). (It seems impossible to copy the data in $o.a_\Rightarrow$ to $o$, although we do not know its formal proof.) Method copy$[a_1, a_2]$ defined in Fig. 21 copies the Boolean-value represented by $o.a_1$ to $o.a_\Rightarrow.a_2$ when it is invoked on $o$. Thus we can obtain the next ID, and the place where the ID is stored is "shifted to right" (see Fig. 20(c), where $\delta(q,1) = (q',0,\rightarrow)$). Next, we explain attribute $a_{\text{cont}}$. This attribute indicates whether the simulation should be continued or not. Let $o$ be the object in which the first element of the current ID is stored. If $o.a_{\text{cont}}$ represents true, then the simulation of $M$ is continued. Otherwise, the simulation stops. For example, in the case of Fig. 20(c), the simulation stops after two steps (Fig. 20(d)). See Lemmas 7 and 8 in Appendix A for a formal description of the behavior of step.

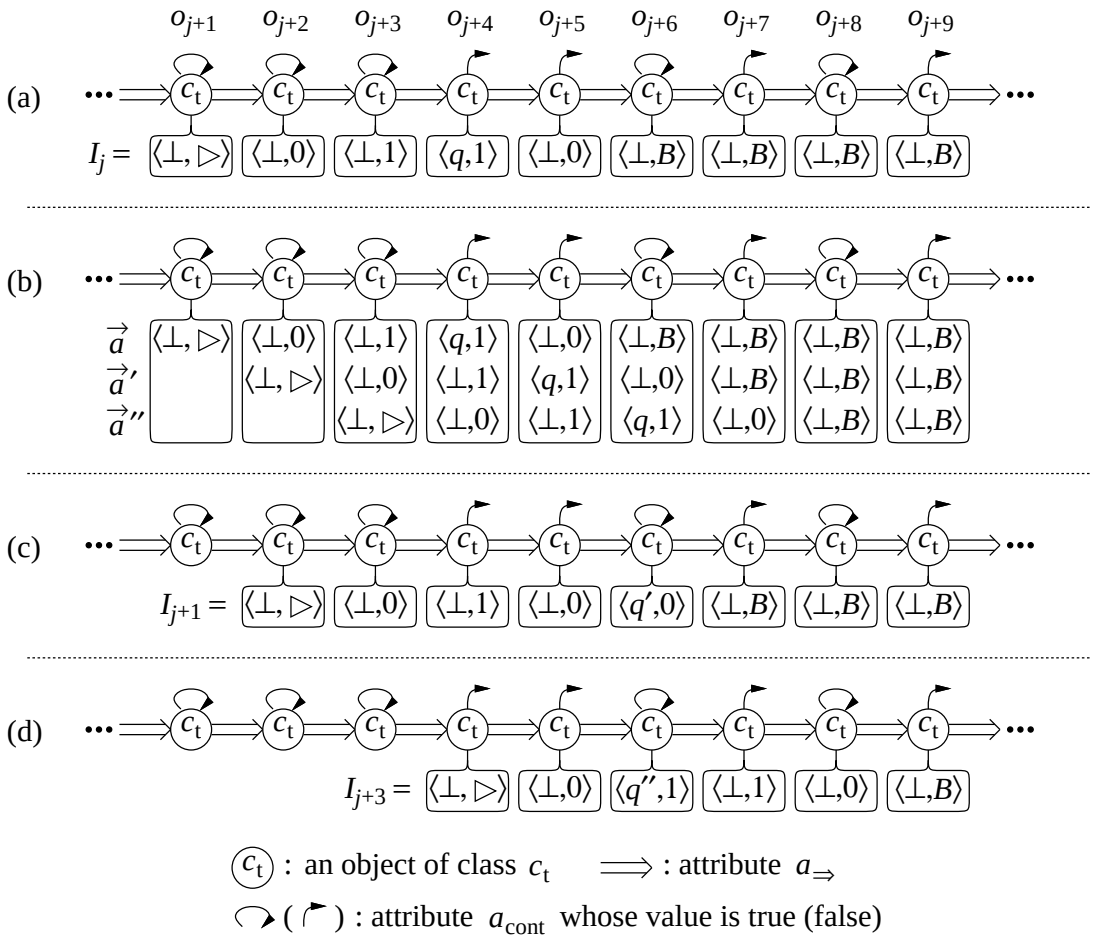



Fig. 20 Working Space to Simulate $M$.

Method accept checks whether $q_{\text{yes}}$ is in the last ID by using nor$[*,*]$ and copy$[*,*]$. It returns $o_{k+1} \in \nu(c'_{\text{t}})$ if $q_{\text{yes}}$ is in the last ID, and $o_{k+1}.a_{\text{f}} \in \nu(c_{\text{f}})$ otherwise. See Lemma 9 in Appendix A for a formal description of the behavior of accept.

Method test is invoked on the returned value of accept. Since test is defined only at class $c_{\text{f}}$, this invocation causes a type error if and only if $q_{\text{yes}}$ is in the last ID.

Suppose that $M$ accepts $x$. Then, $M$ halts after finite steps. Therefore, there is a database instance **I** such that both $k$ and $r$ are large enough to cause a type error under **I**. Conversely, suppose that $M$ does not accept $x$. Since $q_{\text{yes}}$ never appears in the $a_\Rightarrow$-chain, invocation of test causes no type error. Thus, Theorem 3 has been proved.

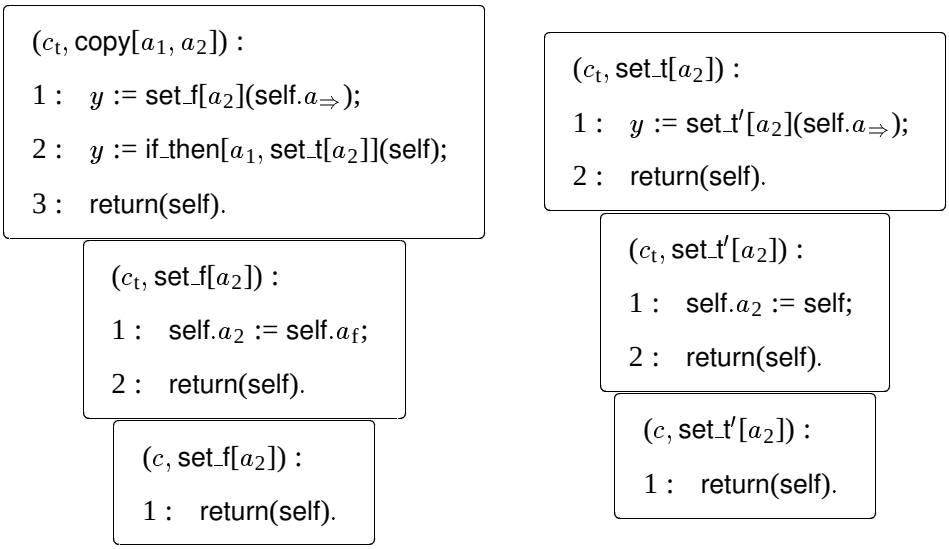



Fig. 21 Definition of Method copy[$a_1$, $a_2$].

In contrast to the above result, consistency for a recursion-free schema with update operations is coNEXPTIME-complete.

**Theorem 4:** Let **S** = ($C, \leq, Attr, Ad, Meth, Impl$) be a recursion-free schema with update operations. Then, consistency for **S** is in coNEXPTIME.

*Proof*: Since **S** is recursion-free, the length of any execution under any instance of **S** is bounded by $N^{|Meth|}$, where $N$ is the maximum number of sentences of a method in *Impl*. Therefore, to find inconsistency for **S**, nondeterministically guess an instance of size at most $N^{|Meth|} \leq \|\mathbf{S}\|^{\|\mathbf{S}\|} = 2^{\|\mathbf{S}\| \log \|\mathbf{S}\|}$ which causes a type error. That is, consistency for **S** is in coNEXPTIME. □

**Theorem 5:** Let **S** = ($C, \leq, Attr, Ad, Meth, Impl$) be a recursion-free schema with update operations. Consistency for **S** is coNEXPTIME-hard, even if the height of $\leq$ is one and *Ad* is covariant.

*Sketch of Proof*: Let $M$ be a fixed $2^{p(n)}$-time bounded nondeterministic Turing machine for a polynomial $p$, and let $x$ be an input string for $M$ with length $n$. We construct, in polynomial time $p'(n)$ of $n$, a recursion-free schema that is inconsistent if and only if $M$ accepts $x$.

The idea of simulating $M$ on $x$ is similar to Theorem 3. However, two problems still remain. First, we have to simulate a nondeterministic transition of $M$. To do this, we introduce new

attributes for each object in the $a_{\Rightarrow}$-chain. The $j$-th nondeterministic choice $ch_j$ is represented by the new attributes of object $o$ in which the first element of the $(j-1)$-th ID $I_{j-1}$ is stored. Then we can compute $I_j[i]$ from $I_{j-1}[i-1]$, $I_{j-1}[i]$, $I_{j-1}[i+1]$, and $ch_j$.

The other problem is how to simulate $2^{p(n)}$ steps of $M$ with a recursion-free schema containing at most $p'(n)$ methods. To solve this problem, we use methods $\mathsf{step}_i$ $(0 \leq i \leq p(n))$ defined as follows:

$(c_{\mathrm{t}}, \mathsf{step}_i) \quad (1 \leq i \leq p(n))$ :
$y$ := $\mathsf{step}_{i-1}(\mathsf{self})$;
$y$ := $\mathsf{step}_{i-1}(y)$;
return($y$).

$(c_{\mathrm{t}}, \mathsf{step}_0)$ :
Simulate one-step transition of $M$;
return(self.$a_{\Rightarrow}$).

It is easily verified that if $\mathsf{step}_{p(n)}$ is invoked on an object $o$ in the $a_{\Rightarrow}$-chain, then $\mathsf{step}_0$ is sequentially invoked on the first $2^{p(n)}$ objects in the $a_{\Rightarrow}$-chain from $o$. A method which simulates one-step transition is defined in the same way since it has to access $2^{p(n)+1}$ objects in the working space. Thus, $2^{p(n)}$ steps of $M$ are simulated by executing $\mathsf{step}_{p(n)}$. The other recursively-defined methods (such as get_ws and accept) in the proof of Theorem 3 are also implemented in the same manner. □

### 3.3 Update Operations

The following theorem can be obtained from Theorem 2 of [16]:

**Theorem 6:** Let **S** = ($C, \leq, Attr, Ad, Meth, Impl$) be a schema that is terminating. If **S** is retrieval, then consistency for **S** is solvable in polynomial time. □

By Theorems 3 and 6, we can conclude that update operations make the type-consistency problem difficult if a given schema is terminating.

## 4. Conclusions

We have discussed the complexity of the type-consistency problem for some subclasses of OODB schemas. Moreover, by comparing the results, we have shown how the complexity is affected by non-flatness of the class hierarchy, recursion, and update operations.

When we classify OODB schemas in view of non-flatness, recursion, and update operations, the type-consistency problem is undecidable or intractable for most of practical OODB schemas. Therefore, as future works, it is desirable to find another subclass of OODB schemas which is practical and for which consistency is tractable. For example, consistency is expected to be decidable for *acyclic* database schemas [12], which are considered as an object-oriented extension of nested relational database schemas. It is also important to develop an incremental algorithm for type-consistency checking.

## References


1. S. Abiteboul, R. Hull and V. Vianu, "Foundations of Databases," Addison-Wesley, 1995.

2. S. Abiteboul, P. Kanellakis, S. Ramaswamy and E. Waller, Method schemas, *J. Comput. System Sci.* **51**, No. 3 (1995), 433–455.

3. S. Abiteboul, P. Kanellaskis and E. Waller, Method Schemas, *in* "Proc. 9th ACM Symposium on Principles of Database Systems," pp. 16–27, 1990.

4. R. Agrawal, L. DeMichiel and B. Lindsay, Static type checking of multi-methods, *in* "Proc. 6th Conf. on Object-Oriented Programming Systems, Languages, and Applications," pp. 113–128, 1991.

5. R. Ahad, J. Davis, S. Gower, P. Lyngbaek, A. Marynowski and E. Onuegbe, Supporting access control in an object-oriented database language, *in* "Proc. 3rd International Conf. on Extending Database Technology," LNCS 580, pp. 184–200, Springer-Verlag, 1992.

6. E. Amiel, M.-J. Bellosta, E. Dujardin and E. Simon, "Type-safe relaxing of schema consistency rules for flexible modelling in OODBMS," *The VLDB Journal* **5**, No. 2 (1996), 133–150.

7. E. Bertino, Data hiding and security in object-oriented databases, *in* "Proc. 8th IEEE International Conf. on Data Engineering," pp. 338–247, 1992.

8. C. Chambers and G. T. Leavens, Typechecking and modules for multimethods, *ACM Trans. on Programming Languages and Systems* **17**, No. 6 (1995), 805–843.

9. J. Eifrig, S. Smith, V. Trifonov and A. Zwarico, Application of OOP type theory: state, decidability, integration, *in* "Proc. 9th Conf. on Object-Oriented Programming Systems, Languages, and Applications," pp. 16–30, 1994.

10. G. Ghelli, A static type system for message passing, *in* "Proc. 6th Conf. on Object-Oriented Programming Systems, Languages, and Applications," pp. 129–145, 1991.

11. R. Hull, K. Tanaka and M. Yoshikawa, Behavior analysis of object-oriented databases: method structure, execution trees, and reachability, *in* "Proc. 3rd International Conf. on Foundations of Data Organization and Algorithms," pp. 372–388, 1989.

12. Y. Ishihara, H. Seki and M. Ito, Type-consistency problems for queries in object-oriented databases, *in* "Proc. 6th International Conf. on Database Theory," LNCS 1186, pp. 364–378, Springer-Verlag, 1997.

13. N. Oxh$\phi$j, J. Palsberg and M. I. Schwartzbach, Making type inference practical, *in* "Proc. European Conf. on Object-Oriented Programming," LNCS 615, pp. 329–349, Springer-Verlag, 1992.

14. J. Palsberg and M. I. Schwartzbach, Object-oriented type inference, *in* "Proc. 6th Conf. on Object-Oriented Programming Systems, Languages, and Applications," pp. 146–161, 1991.

15. J. Palsberg and M. I. Schwartzbach, "Object-Oriented Type Systems," John Wiley & Sons, 1994.

16. H. Seki, Y. Ishihara and M. Ito, Authorization analysis of queries in object-oriented databases, *in* "Proc. 4th International Conf. on Deductive and Object-Oriented Databases," LNCS 1013, pp. 521–538, Springer-Verlag, 1995.

17. E. Waller, Schema updates and consistency, *in* "Proc. 2nd International Conf. on Deductive and Object-Oriented Databases," LNCS 566, pp. 167–188, Springer-Verlag, 1991.

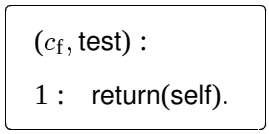



Fig. 22 Definition of Method test.

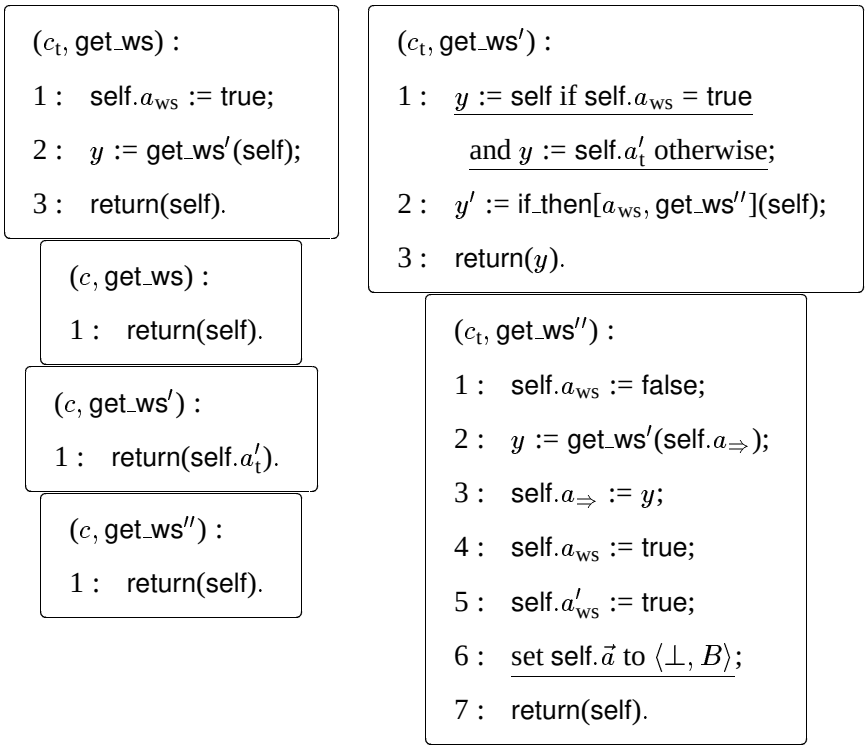



Fig. 23 Definition of Method get_ws.

## Appendix A: Complete Proof of Theorem 3

Let $M$ be a Turing machine and $x = x_1 \cdots x_n$ an input string for $M$. We abbreviate self.$a$ := self and self.$a$ := self.$a_{\mathrm{f}}$ to self.$a$ := true and self.$a$ := false, respectively. Methods test, get_ws, initws, step, accept are defined as shown in Figs. 22–26, respectively.

First, we show that $\mathbf{S}_{M,x}$ is terminating.

**Lemma 3:** Let $\mathbf{I} = (\nu, \mu)$ be an arbitrary database instance of $\mathbf{S}_{M,x}$, and $o_1$ be an arbitrary object in $O_{\mathbf{S}_{M,x},\mathbf{I}}$. The execution of get_ws for $o_1$ is terminating under $\mathbf{I}$.

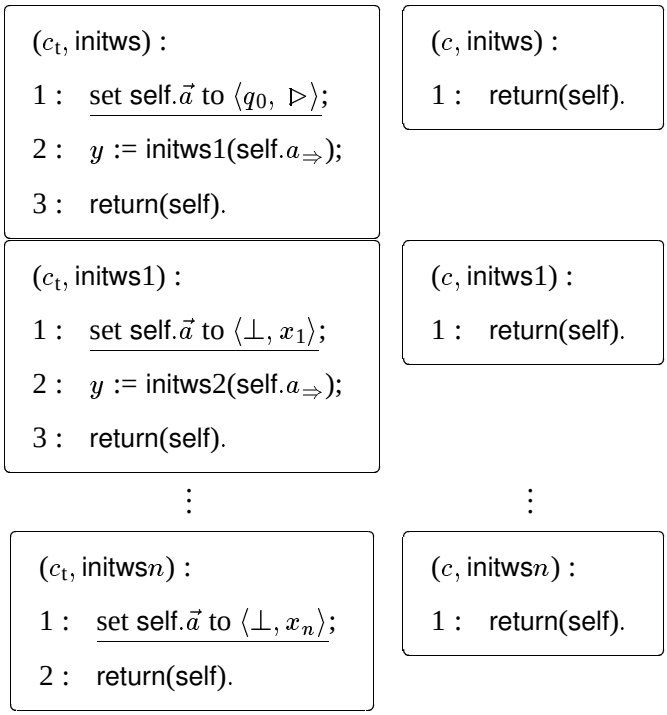



Fig. 24 Definition of Method initws.

*Proof*: If $o_1 \in \nu(c'_t) \cup \nu(c_f) \cup \nu(c)$, then the execution is terminating since ($c$, get_ws) is executed for $o_1$. Thus in the following we consider the remaining case such that $o_1 \in \nu(c_t)$. First of all, by the first line of ($c_t$, get_ws), $o_1.a_{ws}$ is set to true. Then, get_ws′ is invoked on $o_1$. By the second line of ($c_t$, get_ws′), get_ws″ is invoked on $o_1$ since $o_1.a_{ws}$ is true. By ($c_t$, get_ws″), get_ws″ sets $o_1.a_{ws}$ false and recursively invokes get_ws′ on $o_1.a_\Rightarrow$.

Consider the case that get_ws′ is recursively invoked on an object $o$. There are three cases to be considered:

(1) If $o \in \nu(c'_t) \cup \nu(c_f) \cup \nu(c)$, then the recursive invocation of get_ws′ terminates since ($c$, get_ws′) is executed for $o$.

(2) If $o \in \nu(c_t)$ and $o.a_{ws}$ is false, then no more recursive invocation occurs from the definition of ($c_t$, get_ws′).

(3) If $o \in \nu(c_t)$ and $o.a_{ws}$ is true, then get_ws″ is invoked on $o$ by the second line of ($c_t$, get_ws′). Method get_ws″ sets $o.a_{ws}$ false and recursively invokes get_ws′ on $o.a_\Rightarrow$.

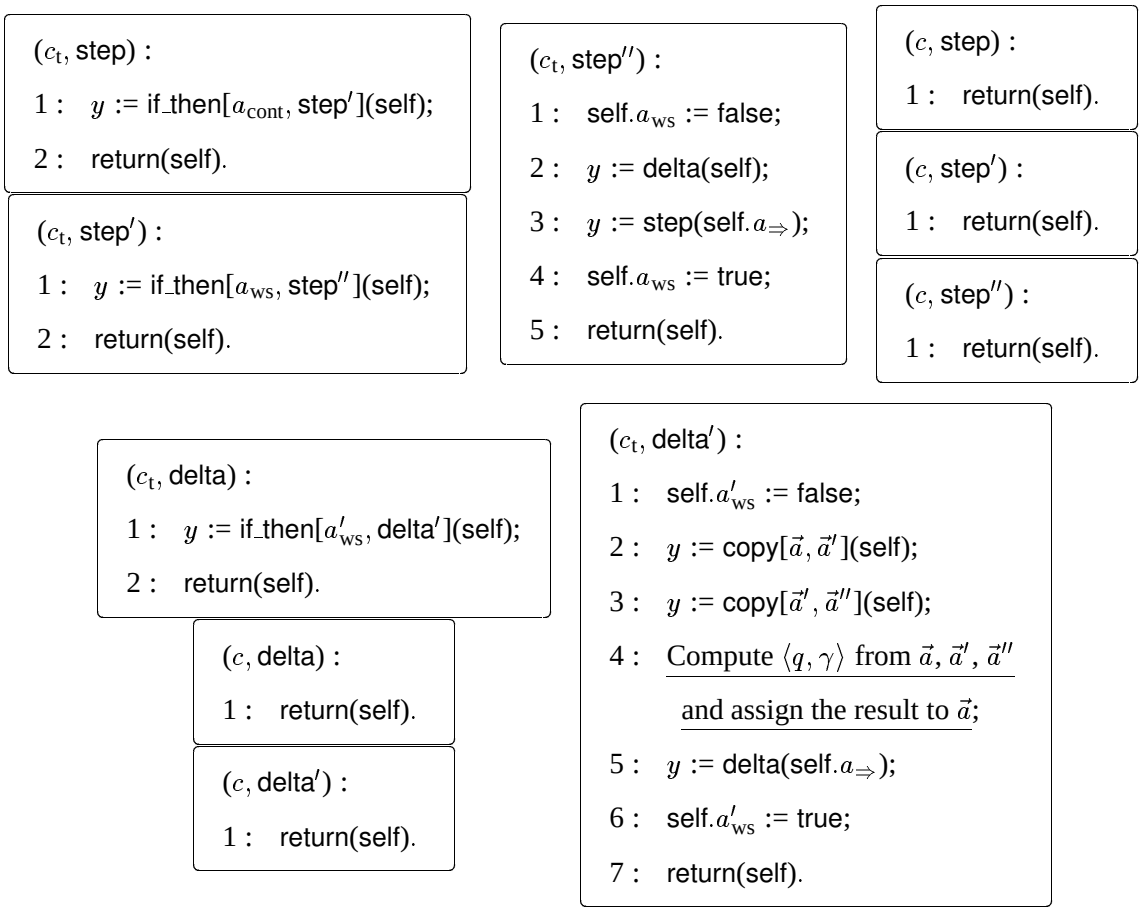



Fig. 25 Definition of Methods step and delta.

Thus, every time get_ws′ is recursively invoked, the number of objects $o$ such that $o.a_{\rm ws}$ is true decreases. Since $O_{\mathbf{S}_{M,x},\mathbf{I}}$ is finite, one of the conditions (1) and (2) above holds eventually.

Therefore, the execution of get_ws on $o_1$ is terminating. □

Similarly, it can be proved that the execution of every recursively-defined method (such as step, delta, accept, etc.) in $\mathbf{S}_{M,x}$ is terminating. Thus we have the following lemma:

**Lemma 4:** $\mathbf{S}_{M,x}$ is terminating. □

In what follows, we show that TM simulates $M$ on $x$ correctly. Hereafter, we mean $o.a$ = $o$ by $o.a$ = true.

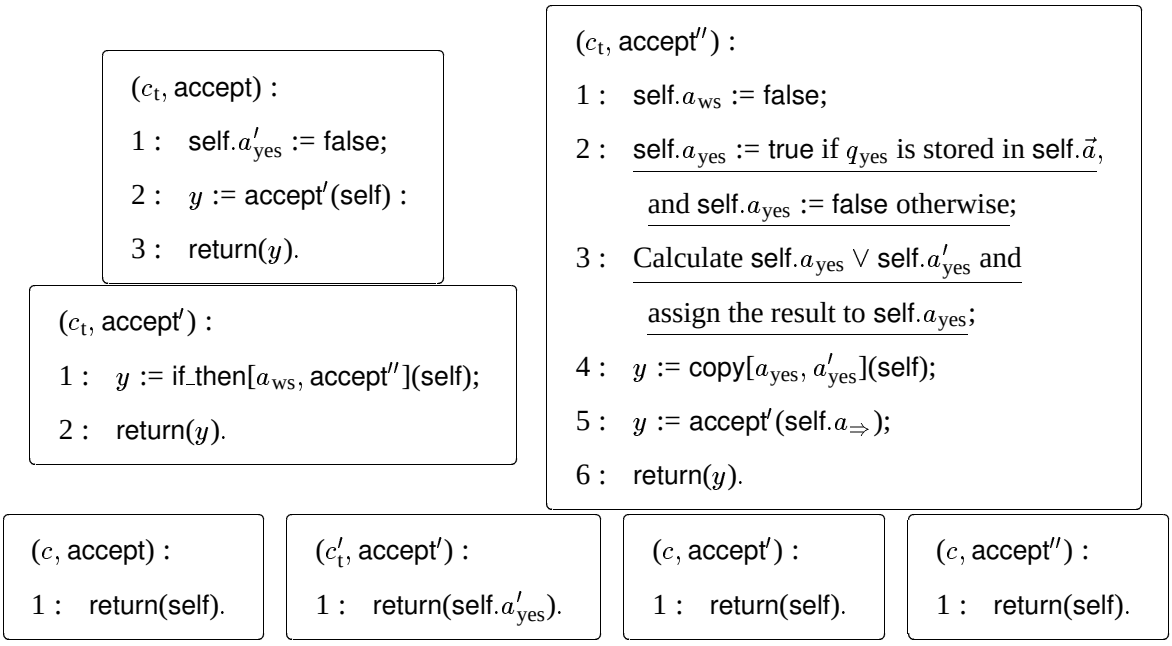



Fig. 26 Definition of Method accept.

**Lemma 5:** Let **I** = ($\nu$, $\mu$) be an arbitrary database instance of $\mathbf{S}_{M,\mathcal{A}}$, and $o_1 \in \nu(c_t)$ be an arbitrary object. After the execution of get_ws for $o_1$ under **I**, there exists a positive integer $k$ which satisfies the following condition (C1):

(C1-1) $o_1 \in \nu(c_t)$, $o_i.a_{\Rightarrow} = o_{i+1} \in \nu(c_t)$ $(1 \leq i \leq k-1)$, and $o_k.a_{\Rightarrow} = o_{k+1} \in \nu(c'_t)$;

(C1-2) $o_i.a_{ws} = o_i.a'_{ws}$ = true $(1 \leq i \leq k)$;

(C1-3) $o_i.\vec{a}$ $(1 \leq i \leq k)$ represents $\langle \perp, B \rangle$.

*Proof*: Suppose that get_ws′ is invoked $k$ times by the second line of ($c_t$, get_ws″) during the complete execution of get_ws for $o_1$. In what follows, we show that $k$ satisfies condition (C1).

First, we prove that $k \geq 1$. By the second line of ($c_t$, get_ws), get_ws′ is invoked on $o_1$. Since $o_1.a_{ws}$ is true by the first line of ($c_t$, get_ws), get_ws″ is invoked on $o_1$ by the second line of ($c_t$, get_ws′). Then, by the second line of ($c_t$, get_ws″), get_ws′ is invoked on $o_1.a_{\Rightarrow} = o_2$. Thus $k \geq 1$.

Next, we prove (C1-1). Consider the $i$-th invocation $(1 \leq i < k)$ of get_ws′ from the second line of ($c_t$, get_ws″). Let $o_{i+1}$ be the self object of the invocation. Note that $o_{i+1} \in \nu(c_t)$ and

$o_{i+1}.a_{\mathrm{ws}}$ is true since $i < k$ (see the condition (3) in the proof of Lemma 3). By the first and third lines of ($c_{\mathrm{t}}$, get_ws′), the returned value of this invocation is $o_{i+1}$. Therefore, by the second and third lines of ($c_{\mathrm{t}}$, get_ws″), it holds that $o_i.a_{\Rightarrow} = o_{i+1} \in \nu(c_{\mathrm{t}})$. Next, consider the $k$-th invocation of get_ws′, and let $o$ be the self object of the invocation. In this case, one of the conditions (1) and (2) in the proof of Lemma 3 holds. If (1) holds, then $o.a'_{\mathrm{t}}$ is returned as the returned value of this invocation since ($c$, get_ws′) is executed for $o$ (see also Fig. 19(1)). If (2) holds, then $o.a'_{\mathrm{t}}$ is returned by the first and third lines of ($c_{\mathrm{t}}$, get_ws′) (see also Fig. 19(2)). Thus, in either case, $o.a'_{\mathrm{t}} \in \nu(c'_{\mathrm{t}})$ is returned and assigned to $o_k.a_{\Rightarrow}$ by the third line of ($c_{\mathrm{t}}$, get_ws″). By letting $o_{k+1}$ be $o.a'_{\mathrm{t}}$, condition (C1-1) is satisfied.

Conditions (C1-2) and (C1-3) hold by the fourth, fifth, and sixth lines of ($c_{\mathrm{t}}$, get_ws″). □

The following lemma holds evidently from the definition of method initws (see Fig. 24).

**Lemma 6:** Suppose that **I** = ($\nu$, $\mu$) satisfies condition (C1) for some $k$ ($k \geq 1$). Then, after the execution of initws for $o_1$ under **I**, the following condition (C2) holds:

(C2-1) The same as (C1-1);

(C2-2) The same as (C1-2);

(C2-3) For each $i$ ($1 \leq i \leq k$), $o_i.\vec{a}$ represents the $i$-th element $I_0[i]$ of the initial ID of $M$ on $x$. □

The following lemma, which states the behavior of method delta (see Fig. 25), is also easily obtained from the explanation in Section 3.2. Intuitively, it states that delta computes a one-step transition of $M$ correctly.

**Lemma 7:** Suppose that **I** = ($\nu$, $\mu$) satisfies the following condition (C3) for some $k$ ($k \geq 1$):

(C3-1) The same as (C2-1);

(C3-2) $o_i.a'_{\mathrm{ws}}$ = true ($1 \leq i \leq k$);

(C3-3) There exists $j$ ($0 \leq j \leq k-1$) such that for each $i$ ($1 \leq i \leq k-j$), $o_{j+i}.\vec{a}$ represents $I_j[i]$.

Then, after the execution of delta for $o_{j+1}$ under **I**, the following condition (C3′) holds:

(C3′-1) The same as (C3-1);

(C3′-2) The same as (C3-2);

(C3′-3) For each $i$ ($1 \leq i \leq k - (j + 1)$), $o_{(j+1)+i}.\vec{a}$ represents $I_{j+1}[i]$. □

**Lemma 8:** Suppose that **I** = ($\nu$, $\mu$) satisfies condition (C2) for some $k$ ($k \geq 1$). Then, after the execution of step for $o_1$ under **I**, the following condition (C4) holds:

(C4-1) The same as (C2-1);

(C4-2) The same as (C2-2);

(C4-3) Let $r$ be the largest index such that for each $l$ ($1 \leq l \leq r$), $o_l.a_{\text{cont}}$ is true, i.e., $r = \max(\{0\} \cup \{j \mid \bigwedge_{l=1}^{j}(o_l.a_{\text{cont}} = o_l)\})$. Then, for each $i$ ($1 \leq i \leq k - r$), $o_{r+i}.\vec{a}$ represents $I_r[i]$.

*Proof*: From the definition of methods step and delta, the value of $o_i.a_{\Rightarrow}$ ($1 \leq i \leq k$) is never altered. Thus, (C4-1) holds by the assumption (C2-1).

Next, we show that (C4-3) is satisfied. By (C2-2), $o_i.a_{\text{ws}}$ is true for each $i$ ($1 \leq i \leq k$). Therefore, by the definitions of ($c_{\text{t}}$, step), ($c_{\text{t}}$, step′), and ($c_{\text{t}}$, step″), it is easily verified that delta is sequentially invoked on $o_1$,..., $o_r$ during the execution of step for $o_1$. Moreover, we claim that:

- (C2) implies (C3) since (C2-3) is obtained by letting $j$ = 0 in (C3-3); and
- (C3′) implies (C3) since (C3-3) is obtained by replacing $j$ + 1 in (C3′-3) by $j$.

Since step can alter $o_i.\vec{a}$ and $o_i.a'_{\text{ws}}$ only by invoking delta, Lemma 7 can be applied $r$ times. Consequently, after the execution of step for $o_1$ under **I**, $o_{r+i}.\vec{a}$ represents $I_r[i]$ for each $i$ ($1 \leq i \leq k - r$). That is, (C4-3) holds.

Lastly, (C4-2) is satisfied because of (C3′-2) and the fourth line of ($c_{\text{t}}$, step″). □

**Lemma 9:** Suppose that $\mathbf{I} = (\nu, \mu)$ satisfies condition (C4) for some $k$ ($k \geq 1$). Then, the returned value of the execution of accept for $o_1$ under $\mathbf{I}$ is $o_{k+1}$ if there is some object $o_i$ ($1 \leq i \leq k$) such that $o_i.\vec{a}$ contains the accepting state $q_{\text{yes}}$, and $o_{k+1}.a_{\text{f}}$ otherwise.

*Proof*: By the first line of ($c_{\text{t}}$, accept), $o_1.a'_{\text{yes}}$ is set to false (i.e., $o_1.a_{\text{f}}$). Then, accept$'$ is invoked on $o_1$. Since $o_1.a_{\text{ws}}$ is true by (C4-2), accept$''$ is invoked on $o_1$. Inductively, consider the execution of accept$''$ for $o_j$ ($1 \leq j \leq k$). By the second line of ($c_{\text{t}}$, accept$''$), $o_j.a_{\text{yes}}$ is set to true (i.e., $o_j$) if $o_j.\vec{a}$ contains $q_{\text{yes}}$, and false (i.e., $o_j.a_{\text{f}}$) otherwise. By the third and fourth lines, $o_{j+1}.a'_{\text{yes}}$ is set to $o_j.a_{\text{yes}} \vee o_j.a'_{\text{yes}}$. Therefore, by the inductive hypothesis, $o_{j+1}.a'_{\text{yes}}$ is set to true (i.e., $o_{j+1}$) if there is some object $o_i$ ($1 \leq i \leq j$) such that $o_i.\vec{a}$ contains $q_{\text{yes}}$, and $o_{j+1}.a'_{\text{yes}}$ is set to false (i.e., $o_{j+1}.a_{\text{f}}$) otherwise.

Lastly, since $o_{k+1} \in \nu(c'_{\text{t}})$ by condition (C4-1), ($c'_{\text{t}}$, accept$'$) is executed for $o_{k+1}$. Therefore, the returned value of the execution of accept for $o_1$ is $o_{k+1}.a'_{\text{yes}}$. Thus, the lemma holds. □

By Lemmas 5–9 and the explanation in Section 3.2, the following lemma holds.

**Lemma 10:** $\mathbf{S}_{M,x}$ is inconsistent if and only if $M$ accepts $x$. □

Theorem 3 is obtained by Lemmas 4 and 10.