オブジェクト指向データベースにおける質問の型検査問題

石原 靖哲　関 浩之　伊藤 実

奈良先端科学技術大学院大学　情報科学研究科

オブジェクト指向データベース (OODB) における質問の型検査問題の計算複雑さについて述べる. OODB スキーマ $\mathbf{S}$ とそのインスタンス $\mathcal{I}$ が与えられたとき, $\mathbf{S}$ 中の各メソッド m の実行中に結合すべきメソッドが常に一意に定まるならば, $\mathcal{I}$ は $\mathbf{S}$ のもとで型整合性をもつという. 本稿では, Hull らによって提案された更新スキーマを OODB スキーマのモデルとして採用し, 与えられた再帰なし更新スキーマのもとで (1) 型整合性をもたないインスタンスが存在するか否かおよび (2) 型整合性をもたない無閉路インスタンスが存在するか否かを判定する問題がいずれも非決定性指数時間完全であることを示す. さらに, (3) 与えられた任意の更新スキーマのもとで型整合性をもたない無閉路インスタンスが存在するか否かを判定する問題が決定不能であることを示す.

Type Consistency Problems for Queries in Object-Oriented Databases

Yasunori ISHIHARA　Hiroyuki SEKI　Minoru ITO

Graduate School of Information Science

Nara Institute of Science and Technology

This paper discusses the computational complexity of type consistency problems for queries in object-oriented databases (OODBs). A database instance is said to be consistent under a database schema if, for every method invocation m, the definition of m to be bound is uniquely determined. In this paper, we adopt update schemas introduced by Hull et al. as a model of OODB schemas, and show that (1) the problem of determining whether there exists an inconsistent instance under a given *recursion-free* update schema and (2) the problem of determining whether there exists an inconsistent *acyclic* instance under a given *recursion-free* update schema are both NEXPTIME-complete. It is also shown that (3) the problem of determining whether there exists an inconsistent *acyclic* instance under a given *arbitrary* update schema is undecidable.

1 Introduction

Among many features of object-oriented programming languages, 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 a method m is applied to an object o, one of its definitions is selected depending on the class to which o belongs, and is bound to m in run-time (late binding or dynamic binding).

This paper discusses the computational complexity of type consistency problems for queries in object-oriented databases (OODBs). A database instance is said to be consistent under a database schema if, for every method invocation m, the definition of m to be bound is uniquely determined by using the class hierarchy with inheritance.

Abiteboul et al. [1] introduced *method schemas*, which correspond to a model of OODB schemas without updating database instances. In Ref. [1], it is shown that

1. the problem of determining whether there exists an inconsistent instance under a given method schema is undecidable in general,
2. NP-complete if every method is recursion-free.

By Dodo et al. [2], it is shown that the problem of determining whether there exists an inconsistent instance under a given monadic method schema (i.e., every method in the schema has at most one argument) is solvable in polynomial time.

On the other hand, Hull et al. [3] introduced *update schemas*, in which updating database instances is simply modeled as assignment of objects or basic values to attributes of objects. Every method in update schemas is monadic. In Ref. [3], it is shown that the problem of determining whether there exists an inconsistent instance under a given update schema is undecidable. In Ref. [4], a subclass of update schemas, called *non-branching update schemas*, is introduced. And, it is shown that the problem of determining whether there exists an inconsistent acyclic instance under a given non-branching update schema is solvable in polynomial time.

Update schemas have all of the basic features of OODBs such as class hierarchy, inheritance, complex objects, and so on. In this paper, we adopt update schemas as a model of OODB schemas, and show that

1. the problem of determining whether there exists an inconsistent instance under a given *recursion-free* update schema is NEXPTIME-complete,
2. the problem of determining whether there exists an inconsistent *acyclic* instance under a given *recursion-free* update schema is also NEXPTIME-complete, and
3. the problem of determining whether there exists an inconsistent *acyclic* instance under a given *arbitrary* update schema is undecidable (see Table 1).

Table 1: Complexity of type consistency problems.

	Instance	
	Acyclic	Arbitrary
Recursion-Free update schema	NEXPTIME-complete†	NEXPTIME-complete†
Arbitrary update schema	Undecidable†	Undecidable

†: Results of this paper.

2 Definitions

2.1 Syntax of Database Schemas

A *database schema* is a 4-tuple $S = (C, \leq, Ad, Impl)$ 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'. We assume that the class hierarchy is a forest on C, that is, for any $c_1, c_2, c \in C$, the following condition is satisfied:

   If $c \leq c_1$ and $c \leq c_2$, then $c_1 \leq c_2$ or $c_2 \leq c_1$.
3. $Ad : C \times Attr \rightarrow C$ is a partial function representing *attribute declarations*, where $Attr$ is a finite set of *attribute names*. 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.
4. $Impl : C \times Meth \rightarrow S$ is a partial function representing *method implementations*, where $Meth$ is a finite set of *method names* and S is a set of *well-formed sequence of sentences* 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). Let $s_1; s_2; \cdots; s_n$ $(= \alpha)$ be a sequence of sentences. We say that α is *well-formed* when the following 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 $y' := m'(y'')$ (y'' is a variable, a' is an attribute, and m' is a method).
- Only the last sentence s_n must have the form return(y') for some variable y'. Thus the other sentences $s_1, s_2, \ldots, s_{n-1}$ must be one of types 1 to 5.

Throughout this paper, we often omit temporary variables for readability. For example, we write "$y := m$(self.a)" instead of "$y' :=$ self.a; $y := m(y')$."

2.2 Semantics of Database Schemas

Let $\mathbf{S} = (C, \leq, Ad, Impl)$ be a database schema. The *inherited implementation* of method m at class c, denoted $Impl^*(c, m)$, is defined as a sequence α such that $Impl(c', m) = \alpha$, where c' is the smallest superclass of c (with respect to the partial order $\leq$) at which an implementation of m exists, that is, if $Impl(c'', m) = \alpha$ and $c \leq c''$, then it must hold that $c' \leq c''$. If such an implementation does not exist, then $Impl^*(c, m)$ is undefined. Similarly, the *inherited attribute declaration* of attribute a at class c, denoted $Ad^*(c, a)$, is defined as a class c'' such that $Ad(c', a) = c''$, where c' is the smallest superclass of c at which an attribute declaration of a exists. If such an attribute declaration does not exist, then $Ad^*(c, a)$ is undefined. A *database instance* of $\mathbf{S}$ is a pair $\mathcal{I} = (\nu, \mu)$, where:

1. To each $c \in C$, ν assigns a disjoint, finite set, denoted $\nu(c)$. Each $o \in \nu(c)$ is called an *object* of class c.
2. To each object $o \in \nu(c)$ and each attribute $a \in A$ such that $Ad^*(c, a) = c'$, μ assigns an object, denoted $\mu(o, a)$, that is called the value of attribute a of o. If $Ad^*(c, a) = c'$, then $\mu(c, a)$ must belong to $\nu(c'')$ for some c'' ($c'' \leq c'$).

Hereafter, we mean $\mu(o, a)$ by $o.a$. If any object o in $\mathcal{I}$ satisfies

$$o.a_1.a_2 \ldots a_n \neq o$$

for any attributes $a_1, a_2, \ldots, a_n$, then $\mathcal{I}$ is called *acyclic*.

The *operational semantics* of a database schema $\mathbf{S}$ under a given database instance $\mathcal{I}$ is formally defined by using MET (method execution tree) introduced in [3]. Here, we do not repeat the formal definition. Instead, we briefly explain its intuitive meaning. As stated before, self represents the object on which a method is invoked; it is called a *self* object.

1. The meaning of a sentence $y := y'$ is obvious.
2. $y :=$ self means that the self object is assigned to variable y.
3. $y :=$ self.a means that the value of attribute a of the self object is assigned to y.
4. If the control reaches a sentence $y := m(y')$, then method m is invoked on the object assigned to y' (or, message m is sent to the object assigned to y') and the returned value is assigned to y. Assume that an object o of a class c is assigned to y'. If $Impl^*(c, m) = \alpha$, then o is bound to "self" in α, α is executed, and the "returned value" is assigned to y. If $Impl^*(c, m)$ is undefined, then a run-time type error occurs.
5. Consider a sentence self.$a := y'$, and let o be the object assigned to y' when the control reaches this sentence. Assume that $Ad^*(c, a) = c' \in C$. If o is an object of a class c'' and $c'' \leq c'$, then the value of attribute a of the self object becomes o. Otherwise, a run-time type error occurs.

2.3 Consistency of Database Schemas

Let $\mathbf{S}$ be a database schema, and $\mathcal{I}$ be a database instance of $\mathbf{S}$. We say that I is *consistent* under $\mathbf{S}$ when the following condition holds:

> Let m be an arbitrary method of $\mathbf{S}$ and $o \in \nu(c)$ be an arbitrary object in $\mathcal{I}$. If $Impl^*(c, m)$ is defined, then no type errors occur during the execution of m on o.

If $\mathcal{I}$ is not consistent under $\mathbf{S}$, then we say that $\mathcal{I}$ is *inconsistent* under $\mathbf{S}$.

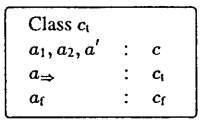


Fig. 1: Ad for basic techniques.

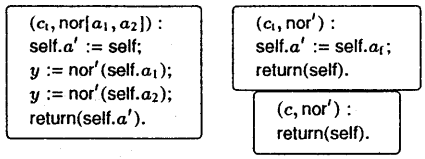


Fig. 2: Methods which calculate NOR of $o.a_1$ and $o.a_2$.

3 Basic Techniques

In this section, we present some basic techniques which are used in the following sections. Throughout this section, C, $\leq$, and Ad are defined as follows:

- $C = \{c, c_t, c_f\}$;
- $\leq$ is the reflexive closure of $\{(c_t, c), (c_f, c)\}$ (i.e., c is a superclass of both c_t and c_f); and
- Ad is shown in Fig. 1.

Let o be an object of class c_t. Each attribute $a \in \{a_1, a_2, a', a_f\}$ of o represents a Boolean value: a represents true if $o.a = o$, and false otherwise. Note that a_f of o always represents false because of the declaration $(c_t, a_f) : c_f$ in Ad.

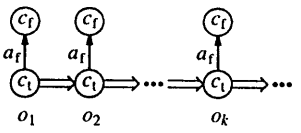
First, we show a method nor in Fig. 2, which calculates NOR of $o.a_1$ and $o.a_2$. Since any Boolean operator can be represented by NORs, we can construct a method which calculates any given Boolean formula by using nor. Formally, we have the following lemma:

Lemma 1: Let o be an object of class c_t. Let o' denote the object returned by the execution of method nor$[a_1, a_2]$ on o. Then, the following equation holds:

$$o' = \begin{cases} o & (\text{if } o.a_1 \neq o \text{ and } o.a_2 \neq o), \\ o.a_f & (\text{otherwise}). \end{cases}$$

Proof: Consider how $o.a'$ changes during the execution of nor$[a_1, a_2]$. First, $o.a'$ is set to o. By the second line of $(c, \text{nor}[a_1, a_2])$, $o.a'$ is set to $o.a_f$ if $o.a_1 = o$, and unchanged otherwise. Similarly, by the third line, $o.a'$ is set to $o.a_f$ if $o.a_2 = o$, and unchanged otherwise. Therefore, $o.a'$ is set to $o.a_f$ if $o.a_1 = o$ or $o.a_2 = o$, and unchanged (i.e., $o.a' = o$) otherwise. □

Next, consider a database instance of this schema shown in Fig. 3. By invoking a method copy (Fig. 4) on



$\Longrightarrow : a_{\Rightarrow}$ $\textcircled{c}$: an object of class c

Fig. 3: A database instance.

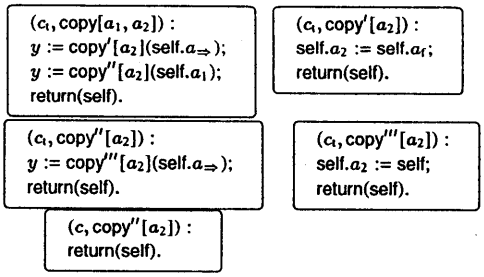


Fig. 4: Methods which copy $o_j.a_1$ to $o_{j+1}.a_2$.

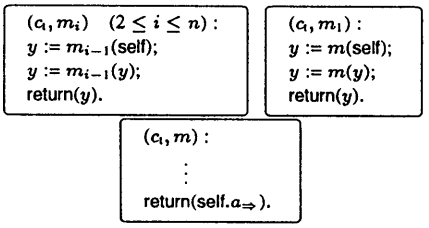


Fig. 5: Method which sequentially invokes m on 2^n objects.

an object o_j in the "$a_{\Rightarrow}$-chain," the Boolean value represented by $o_j.a_1$ is copied to $o_{j+1}.a_2$ ($= o_j.a_{\Rightarrow}.a_2$). Formally, we have the following lemma:

Lemma 2: Let o_j be an object of class c_t. After the execution of method $\text{copy}[a_1, a_2]$ on o_j, the following equation holds:

$$o_{j+1}.a_2 = \begin{cases} o_{j+1} & (\text{if } o_j.a_1 = o_j), \\ o_{j+1}.a_f & (\text{otherwise}). \end{cases}$$

Proof: An easy observation proves this lemma. □

Lastly, see again Fig. 3. Suppose that a method m returns $o.a_{\Rightarrow}$ when m is invoked on o. Define method m_n as shown in Fig. 5. It is easy to see that m_n sequentially invokes m on 2^n objects $o_1, \ldots, o_{2^n}$ in the "$a_{\Rightarrow}$-chain." Note that m_i $(1 \le i \le n)$ can be constructed in polynomial time of n.

4 Recursion-Free Schemas

Definition 1: Problem *RF/AC* is to determine whether there exists an inconsistent acyclic instance under a given recursion-free schema S. □

We show that *RF/AC* is NEXPTIME-complete.

Lemma 3: *RF/AC* is in NEXPTIME.

Proof: Since S is recursion-free, execution of any method in S always terminates and the number of objects traversed during the execution is bounded by $|\mathbf{S}|^{|\mathbf{S}|}$. Therefore, to solve *RF/AC*, nondeterministically guess an instance of size $|\mathbf{S}|^{|\mathbf{S}|} = 2^{|\mathbf{S}| \log |\mathbf{S}|}$ which causes a type error. □

To show that *RF/AC* is NEXPTIME-hard, we reduce any language in NEXPTIME to *RF/AC*. To do this, for a given input string x of a fixed $2^{p(n)}$-time bounded nondeterministic Turing machine M, we construct, in polynomial time of $|x|$, a schema $\mathbf{S}_{M,x}$ such that some acyclic instance is inconsistent under $\mathbf{S}_{M,x}$ if and only if M accepts x. First, we define a nondeterministic Turing machine and an instantaneous description.

Definition 2: A nondeterministic Turing machine M is a triple (Q, Σ, δ), where

- Q is a finite set of states. Q has three special states: the initial state q_0, the accepting state q_{yes}, and the rejecting state q_{no};
- Σ is a finite set of symbols. Σ has 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; and
- δ is a function which maps $(Q - \{q_0, q_{\text{yes}}, q_{\text{no}}\}) \times \Sigma$ to the power set of $Q \times \Sigma \times \{\leftarrow, \rightarrow, -\}$. δ must satisfy the following conditions:
  - For each pair $(q, \sigma) \in (Q - \{q_0, q_{\text{yes}}, q_{\text{no}}\}) \times \Sigma$, $|\delta(q, \sigma)|$ (the number of possible nondeterministic choices) is at most two. Assume that the elements of each $|\delta(q, \sigma)|$ are identified by 0 and 1; and
  - If $(q', \sigma, d) \in \delta(q, \triangleright)$, then $\sigma = \triangleright$ and $d = \rightarrow$. Therefore, the tape head never falls off the left end of the tape. □

Definition 3: An instantaneous description (ID) I of M is a finite sequence $\langle q_1, \sigma_1 \rangle, \ldots, \langle q_k, \sigma_k \rangle$, where $q_i \in Q \cup \{\perp\}$ and $\sigma_i \in \Sigma$. It is required that exactly one q_i is in Q (head position). The i-th pair $\langle q_i, \sigma_i \rangle$ of an ID I is denoted by $I[i]$. The transition relation $\vdash_M$ over the set of ID's are defined as usual. Let I_0 denote an initial ID. □

Let $M = \langle Q, \Sigma, \delta \rangle$ be a $2^{p(n)}$-time bounded nondeterministic Turing machine. Let $x \in (\Sigma - \{B, \triangleright\})^*$ be an input string for M. Let $n = |x|$ and $N = 2^{p(n)}$. Let $K = \lceil \log(|Q| + 1) \rceil + \lceil \log |\Sigma| \rceil$. For M and x, define C, $\le$, and Ad as follows:

- $C = \{c, c_{t0}, c_{t1}, c_{tt}, c_f\}$;
- $\le$ is the smallest partial order such that
  1. $c_{t0} \le c_{t1} \le c_{tt} \le c$, and
  2. $c_f \le c$; and
- Ad is shown in Fig. 6. Strictly speaking, some more temporary attributes are necessary to store intermediate result of calculation.

An example of an acyclic database instance of $\mathbf{S}_{M,x}$ is shown in Fig. 7.

Let I be a given acyclic instance of $\mathbf{S}_{M,x}$. Let o_1 be an object of class c_{t0} or c_{t1} in I. Note that the "$a_{\Rightarrow}$-chain"

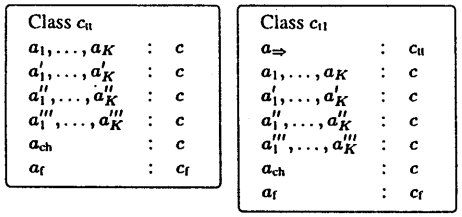


Fig. 6: Ad of $\mathbf{S}_{M,x}$.

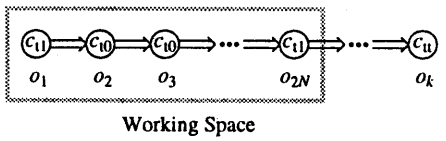


Fig. 7: An acyclic instance of $\mathbf{S}_{M,x}$.

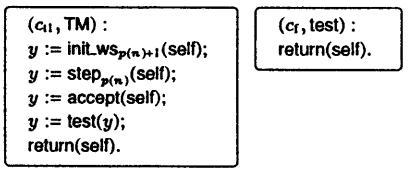


Fig. 8: Methods TM and test.

from o_1 terminates in an object of class c_{t1} (see Figs. 6 and 7). The $a_\Rightarrow$-chain is used as working space to simulate M.

We give an intuitive explanation of the behavior of TM (defined in Fig. 8) invoked on o_1. Let o_j ($1 \leq j \leq k$, $k \geq 1$) be the objects which form the $a_\Rightarrow$-chain, where $o_j.a_\Rightarrow = o_{j+1}$ ($1 \leq j \leq k-1$) and o_k is an object of class c_{t1} (see Fig. 7).

1. Initialize the first $2N = 2^{p(n)+1}$ objects (line 1 of (c_{t1}, TM) of Fig. 8). More precisely, for each j ($1 \leq j \leq 2N$), $I_0[j]$ is stored in $o_j.a_1, \ldots, o_j.a_K$ by binary encoding. The reason why $2N$ objects are initialized as working space is explained later.
2. Rewrite the ID stored in the working space $N = 2^{p(n)}$ times (line 2). We consider that the nondeterministic choice at j-th step of M is given by the class to which object o_j belongs, where c_{t0} corresponds to choice 0 and c_{t1} does to choice 1. Method step is defined below.
3. Check whether the accepting state q_{yes} is in the last ID (line 3). The returned value of accept is an object of class c_{t1} (or its subclass) if q_{yes} is in the last ID. Otherwise, an object of class c_f is returned. Method accept can be easily constructed by using methods nor, copy$[a_1, a_2]$ and m_n stated in Sect. 3.
4. Invoke test on the returned value of the previous step (line 4). Since method test is defined only for class c_f, that will cause a type error if and only if q_{yes} is in the last ID. Therefore, there is some acyclic instance which is inconsistent under $\mathbf{S}_{M,x}$ if and only if M accepts x.

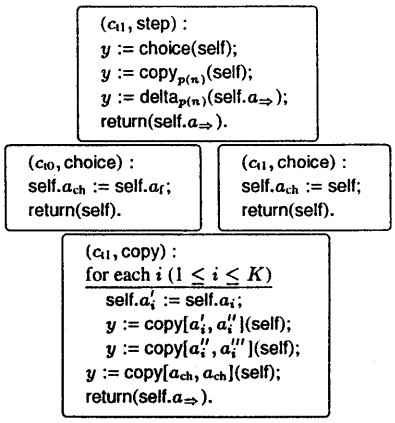


Fig. 9: Methods step, choice, and copy.

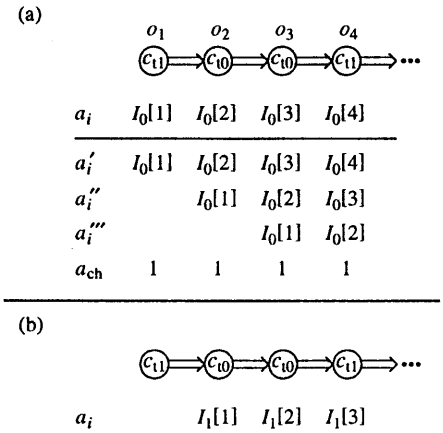


Fig. 10: Rewriting an ID.

We define method step as shown in Fig. 9. To explain the behavior of step, suppose that each $I_0[j]$ is stored in $o_j.a_1, \ldots, o_j.a_K$. Let I_1 be an ID such that $I_0 \vdash_M I_1$. Now suppose that step is invoked on o. By method choice (Fig. 9), the nondeterministic choice $ch_1 \in \{0, 1\}$ at the first step, which is given by the class to which o_1 belongs, is stored in $o_1.a_{ch}$. Then, method copy (Fig. 9) is invoked on each o_j ($1 \leq j \leq N$). The underlined parts (e.g., the first line of (c_{t1}, copy)) are macro notation. All of them can be expanded when M and x are reduced to $\mathbf{S}_{M,x}$. After the invocation of copy, each object o_j has $I_0[j-2]$, $I_0[j-1]$, $I_0[j]$, and ch_1 (see Fig. 10(a)). Next, method delta is invoked on each object o_j ($2 \leq j \leq N+1$) to obtain $I_1[j-1]$, which is to be stored in $a_1, \ldots, a_K$ of o_j (see Fig. 10(b)). Since the position stored the ID in the working space is shifted to right at each step, $2N$ objects are necessary as working space. We can construct delta in constant time with respect to n by using nor stated in Sect. 3.

Now we have the following theorem:

Theorem 1: RF/AC is NEXPTIME-complete. □

Let RF be the problem of determining whether there exists an inconsistent instance under a given recursion-free schema S. By slightly modifying the construction of $S_{M,x}$, we have the following theorem:

Theorem 2: RF is NEXPTIME-complete. □

5 Arbitrary Schemas, Acyclic Instances

Definition 4: Problem AC is to determine whether there exists an inconsistent acyclic instance under a given schema S. □

We prove that AC is undecidable by showing a reduction from any recursively enumerable language to AC. To do this, for a given input string x of a fixed Turing machine M, we construct a schema $S'_{M,x}$ such that an acyclic instance is inconsistent under $S'_{M,x}$ if and only if M accepts x.

Let $M = \langle Q, \Sigma, \delta \rangle$ be a deterministic Turing machine, i.e., for each pair $(q, \sigma) \in (Q - \{q_0, q_{yes}, q_{no}\}) \times \Sigma$, $|\delta(q, \sigma)|$ is at most one. Let $x \in (\Sigma - \{B, \triangleright\})^*$ be an input string for M. Let $K = \lceil \log(|Q| + 1) \rceil + \lceil \log |\Sigma| \rceil$. For M and x, define C, $\leq$, and Ad to be the same as $S_{M,x}$ in Sect. 4.

Let us construct TM′ which simulates M on x. Since recursion is allowed now, we modify init_ws, step, etc. in Sect. 4 so that they recursively traverse the $a_{\Rightarrow}$-chain until it reaches object o_k of class c_{t1}. Therefore, $\text{init\_ws}_{p(n)+1}$, $\text{step}_{p(n)}$, etc. are not necessary. Moreover, it is unknown when M stops in advance. A tentative solution would be as follows:

```
(c_t1, TM′) :
y := init_ws(self);
y := step&accept&test(self);
return(self).

(c_t1, step&accept&test) :
y := step(self);
y := accept(self);
y := test(y);
y := step&accept&test(self.a⇒);
return(self).
```

However, this does not work since step&accept&test may invoke on an object in an $a_{\Rightarrow}$-chain which is not initialized. Hence, it is possible that some acyclic instance is inconsistent under $S'_{M,x}$ even if M does not accept x.

Instead, we define TM′ and step′ as shown in Fig. 11. Classes c_{t0} and c_{t1} represent the choice whether rewriting the ID is continued or not. To explain this more precisely, consider the situation that TM′ is invoked on o_1 in Fig. 7. All the object in the $a_{\Rightarrow}$-chain are initialized by method init_ws. Then step′ is invoked on o_1. If o_1 is of class c_{t0}, then the ID stored in the $a_{\Rightarrow}$-chain is rewritten and step′ is recursively invoked on o_2. This recursive invocation is repeated until some object o_j of class c_{t1} or c_{t1} is encountered. That is, if $o_j \in \nu(c_{t0})$ for each j $(1 \leq j \leq k')$ and $o_{k'} \in \nu(c_{t1}) \cup \nu(c_{t1})$, then TM′ simulates M up to k' steps.

Now we have the following theorem:

Theorem 3: AC is undecidable. □

```
(c_t1, TM′) :
y := init_ws(self);
y := step′(self);
y := accept(self);
y := test(y);
return(self).

(c_t1, step′) :
return(self).

(c_t0, step′) :
y := copy(self);
y := delta(self.a⇒);
y := step′(self.a⇒);
return(self).
```

Fig. 11: Methods TM′ and step′.

Consider executing TM′ on a cyclic database instance. Since all the attributes except $a_{\Rightarrow}$ are initialized by init_ws, we can focus on the case that $a_{\Rightarrow}$ forms a cycle. In this case, TM′ does not terminate. More precisely, init_ws is invoked infinitely many times without type error. Therefore, we have the following known result [3] as a corollary of Theorem 3:

Corollary 1: The problem of determining whether there exists an inconsistent instance under a given schema S is undecidable. □

6 Conclusions

Theorems 1 and 2 mean that there are no algorithms to solve RF/AC or RF better than the obvious algorithm stated in Lemma 3. On the other hand, as stated in Sect. 1, these problems for method schemas are solvable in polynomial time (recall that method schemas do not update database instances). It is interesting to find a subclass of update schemas for which type consistency problems are solvable more efficiently than NEXPTIME.

References

[1] S. Abiteboul, P. C. Kanellakis and E. Waller: "Method Schemas," Proc. 9th ACM Symposium on Principles of Database Systems, pp. 16–27, 1990.

[2] H. Dodo, Y. Ishihara, H. Seki and T. Kasami: "An Algorithm for Testing Consistency of Method Schemas," IPSJ SIG Notes, PRG-15-5, Jan. 1994 (in Japanese).

[3] R. Hull, K. Tanaka and M. Yoshikawa: "Behavior Analysis of Object-Oriented Databases: Method Structure, Execution Trees, and Reachability," Proc. 3rd Int'l Conf. on Foundations of Data Organization and Algorithms, pp. 372–388, June 1989.

[4] H. Seki, Y. Ishihara and M. Ito: "Authorization Analysis of Queries in Object-Oriented Databases," Proc. of the Fourth International Conference on Deductive and Object-Oriented Databases '95 (Singapore), Dec. 1995 (to appear).