NAIST-IS-DT9861011

Doctor's Thesis

Complexity of the Type-Consistency Problem in Object-Oriented Databases

Shougo Shimizu

February 5, 2001

Department of Information Processing
Graduate School of Information Science
Nara Institute of Science and Technology

Doctor's Thesis
submitted to Graduate School of Information Science,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
DOCTOR of ENGINEERING

Shougo Shimizu

Complexity of the Type-Consistency Problem in Object-Oriented Databases*

Shougo Shimizu

Abstract

Method invocation mechanism is one of the essential features in object-oriented programming languages. It is based on method name overloading and dynamic binding by method inheritance along the class hierarchy. For a method name m, different classes may have different definitions of m (method name overloading). When m is applied to an object o, one of its definitions is selected depending on the class to which o belongs and bound to m in runtime (dynamic binding). This mechanism is important for data encapsulation and code reuse, but there is a risk of runtime type errors. For example, when a method m is invoked, the definition of m to be bound may not exist. Particularly, with queries in object-oriented databases (OODBs), a runtime error causes rollback, i.e., all the modification up to the error must be canceled. Therefore, for a given OODB schema, it is desirable to ensure that no runtime type error occurs during the execution of queries under any instance of the OODB schema.

An OODB schema is said to be consistent if no type error occurs during the execution of any method under any database instance. The type-consistency problem is to decide whether a given OODB schema is consistent or not. This thesis discusses the computational complexity of the type-consistency problem.

As a model of OODB schemas, this thesis adopts update schemas. Update schemas have all 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 to attributes of objects.

* Doctor's Thesis, Department of Information Processing, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-DT9861011, February 5, 2001.

The type-consistency problem for update schemas is known to be undecidable in general. Furthermore, the following results of the complexity of the type-consistency problem for subclasses of update schemas are known: The problem is (1) solvable in polynomial time for flat schemas, i.e., schemas that have no class hierarchy, (2) undecidable for non-flat schemas even if the height of the class hierarchy is at most one, (3) solvable in polynomial time for terminating retrieval schemas, and (4) coNEXPTIME-complete for recursion-free schemas.

This thesis shows that the type-consistency problems for terminating schemas and retrieval schemas are both undecidable. From these results, it turns out that recursion and update operations (only when the schema satisfies the termination property), as well as non-flatness of the class hierarchy, each make the problem difficult.

This thesis also introduces a subclass of update schemas, called acyclic schemas, for which the type-consistency problem becomes decidable. A schema is said to be acyclic if no instance of the schema contains a cycle which is formed by attribute-value relationship among objects. A schema which represents an organization of a company can be considered as an example of an acyclic schema. Furthermore, a nested relational model is considered to be a special case of acyclic schemas. This thesis shows the following results of the complexity of the type-consistency problem for acyclic schemas: The problem is (1) in coNEXPTIME for acyclic schemas, (2) coNEXPTIME-hard for recursion-free acyclic schemas, and (3) PSPACE-complete for retrieval acyclic schemas.

Keywords:

complexity, type-consistency problem, object-oriented database, runtime type error, update schema, acyclic schema

Acknowledgments

I have been fortunate to have received support and assistance from many individuals. I would like to thank Professor Minoru Ito for his valuable comments and continuous support throughout the work. I am also grateful to Professor Shunsuke Uemura for his valuable suggestions and discussions on the work. I also would like to thank Professor Hiroyuki Seki for his valuable comments, suggestions, and encouragement throughout the work.

I would like to deeply thank Assistant Professor Yasunori Ishihara of Osaka University for his direction throughout the work. Without his invaluable support, the work could not have been completed.

I would especially like to thank Dr. Yeon-Dae Kwon for her kind support and encouragement throughout the work. Thanks to her, I could have a pleasant life in the laboratory.

I am grateful to Mr. Tomotaka Takarabe and Mr. Junji Yokouchi, who have helped me do the work when they have been the master course students of Nara Institute of Science and Technology. I am also grateful to Dr. Toshiyuki Morita for his kind help and encouragement. I am obliged to all the members of Ito Laboratory of Nara Institute of Science and Technology.

Finally, I sincerely thank my parents for caring very much about me for a long time.

List of Publications

1. Publications Related to the Thesis

Journal Papers

[1] S. Shimizu, Y. Ishihara, H. Seki, and M. Ito, "Complexity of the type-consistency problem in object-oriented databases," IEICE Transactions on Information and Systems, vol.J81-D-I, no.3, pp.261–270, March 1998 (in Japanese).

[2] S. Shimizu, Y. Ishihara, J. Yokouchi, and M. Ito, "Complexity of the type-consistency problem for acyclic object-oriented database schemas," IEICE Transactions on Information and Systems (to appear).

[3] Y. Ishihara, S. Shimizu, H. Seki, and M. Ito, "Refinements of complexity results on type-consistency for object-oriented databases," Journal of Computer and System Sciences (conditionally accepted).

Workshops

[4] S. Shimizu, Y. Ishihara, H. Seki, and M. Ito, "Complexity of the type-consistency problem in object-oriented databases," IEICE Technical Report, COMP97-19, May 1997 (in Japanese).

[5] J. Yokouchi, S. Shimizu, Y. Ishihara, and M. Ito, "Complexity of the type-consistency problem for acyclic object-oriented database schemas," IEICE Technical Report, COMP99-16, June 1999 (in Japanese).

[6] S. Shimizu, J. Yokouchi, Y. Ishihara, and M. Ito, “Complexity of the type-consistency problem for retrieval and recursion-free acyclic schemas,” IEICE Technical Report, COMP99-88, March 2000.

2. Other Publications

Journal Papers

[7] Y.D. Kwon, Y. Ishihara, S. Shimizu, and M. Ito, “Computational complexity of finding highly co-occurrent itemsets in market basket databases,” IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, vol.E83-A, no.12, pp.2723–2735, Dec. 2000.

International Conferences

[8] S. Shimizu, Y. Ishihara, T. Takarabe, and M. Ito, “A probabilistic database model with representability of dependency among tuples,” Proceedings of the 4th World Multiconference on Systemics, Cybernetics and Informatics, volume VIII, pp.221–225, July 2000.

Workshops

[9] T. Takarabe, S. Shimizu, Y. Ishihara, and M. Ito, “A probabilistic database model with representability of dependency among tuples,” IPSJ SIG Notes, 2000-DBS-120, Jan. 2000 (in Japanese).

[10] Y.D. Kwon, S. Shimizu, Y. Ishihara, and M. Ito, “Computational complexity of finding highly co-occurrent itemsets,” IEICE Technical Report, COMP99-89, March 2000.

Contents

List of Figures

Chapter 1

Introduction

Among the 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 dynamic binding by method inheritance along the class hierarchy. For a method name m, different classes may have different definitions (codes, implementations) of m (method name overloading). When 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 runtime (dynamic binding or late binding). This mechanism is important for data encapsulation and code reuse, but there is a risk of runtime type errors. For example, when a method m is invoked, the definition of m to be bound may not exist. Particularly, with queries in object-oriented databases (OODBs), a runtime error causes rollback, i.e., all the modification up to the error must be canceled.

In this thesis, we discuss the computational complexity of the type-consistency problem for queries in OODBs. A database schema $\mathbf{S}$ 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 using the class hierarchy with inheritance; and
2. no attribute-value update violates any type declaration given by $\mathbf{S}$.

The type-consistency problem is to decide whether a given database schema is consistent or not. In order to check type-consistency, it is usually necessary to

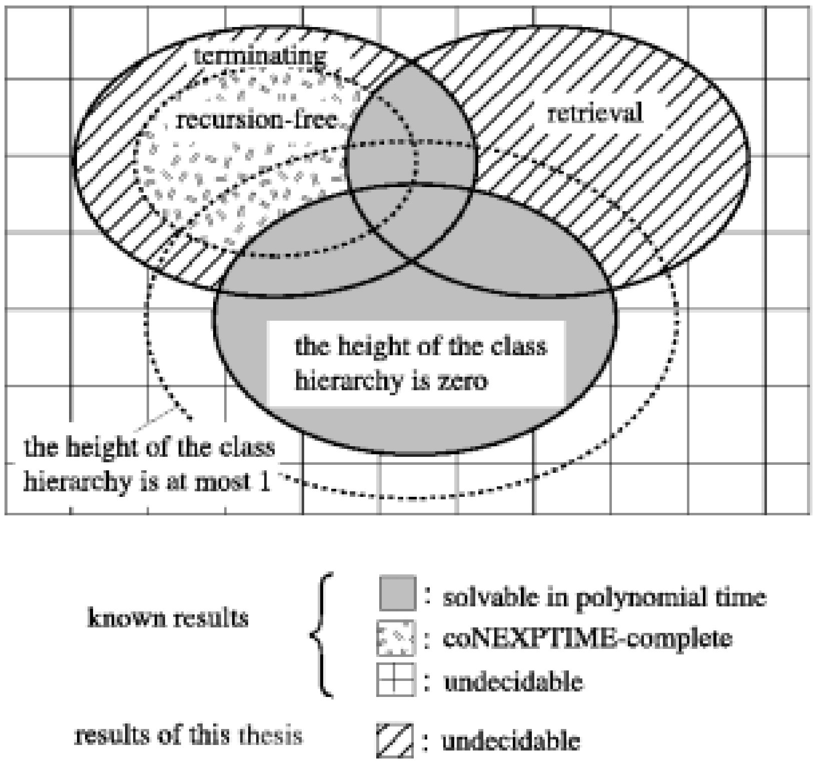

terminating
recursion-free
retrieval
the height of the class hierarchy is zero
the height of the class hierarchy is at most 1
known results
: solvable in polynomial time
: coNEXPTIME-complete
: undecidable
results of this thesis
: undecidable

Figure 1.1. Complexity of the type-consistency problem.

perform type inference, i.e., to examine whether for each class c and program construct x, e.g., 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, runtime type check can be omitted. Another advantage is an application to method-based authorization checking [5, 7, 18].

As a model of OODB schemas, we adopt *update schemas* introduced by Hull et al [11]. Update schemas have all 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 to attributes of objects.

In Reference [11], it is shown that the type-consistency problem for update schemas is undecidable in general. Reference [12] have introduced subclasses of update schemas by the following three factors and shown the following results of the type-consistency problem.

1. *Non-flatness of the class hierarchy.* 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 “flat” schemas, consistency is solvable in polynomial time. However, consistency for a non-flat schema is undecidable even if the height of the class hierarchy is at most one.

2. *Recursion.* Consistency for recursion-free schemas is coNEXPTIME-complete.

3. *Update operations.* Consistency for terminating retrieval schemas is solvable in polynomial time.

This thesis shows the following two results of the type-consistency problem which have been open in Reference [12] (see Figure 1.1).

1. Consistency for non-flat schemas 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 at most one.

2. Consistency for schemas 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 at most one.

From these results, it turns out that recursion and update operations (only when the schema satisfies the termination property), as well as non-flatness of the class hierarchy, each make the problem difficult. When we classify update schemas in view of non-flatness, recursion, and update operations, the type-consistency problem is undecidable or intractable for most of practical update schemas.

This thesis also introduces a subclass of update schemas, called *acyclic schemas*, for which the type-consistency problem becomes decidable. A schema is said to be acyclic if no instance of the schema contains a cycle which is formed by attribute-value relationship among objects. A schema which represents an organization of a company can be considered as an example of an acyclic schema.

Furthermore, a nested relational model [1] is considered to be a special case of acyclic schemas. The followings are the results of this thesis.

1. The type-consistency problem for acyclic schemas is in coNEXPTIME;
2. The problem for *recursion-free acyclic schemas* is coNEXPTIME-hard; and
3. The problem for *retrieval acyclic schemas* is PSPACE-complete.

The following three restrictions are placed on the model of OODB schemas in this thesis. First, every method should be monadic (i.e., every method in a schema should be unary). Thus, only a target object can be specified as an argument of each method. Even if the arity is not bounded, consistency is expected to be still decidable for a flat schema, a recursion-free schema, a terminating retrieval schema, and an acyclic schema respectively. That is, in our conjecture, arity does not affect the decidability of consistency as long as we consider only the subclasses of schemas stated above. Second, this model has no program constructs such as conditional branch and iterative statement. However, they can be simulated in this model, as will be shown in Section 2.4. Thirdly, the class hierarchy should be a forest (i.e., multiple inheritance is excluded). However, the results in this thesis remain valid if an appropriate mechanism for multiple inheritance is incorporated into the model. That is, the third restriction is merely for simplicity.

Related Work

There has been much research on the type-consistency problem for OOPLs. For example, Abiteboul et al. have introduced *method schemas* and studied the complexity of the type-consistency problem for many subclasses of them [2]. In method schemas, each method is allowed to have more than one arguments. 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 Reference [2].

1. The type-consistency problem for method schemas is undecidable in general;
2. The decidability is open for a method schema with methods of arity at most two;

3. The problem for recursion-free method schemas is coNP-complete; and

4. The problem for monadic method schemas, a proper subclass of retrieval schemas of ours, is solvable in polynomial time.

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 at most two, and their intersection is not empty. In this thesis, we prove the undecidability for a retrieval schema which belongs to the intersection. That is, the open problem 2 above is shown to be undecidable. For more details, refer to the end of Section 3.1, where we briefly discuss the idea of how to translate a monadic retrieval schema into a method schema of arity at most two. In Reference [19], an optimal incremental algorithm for the consistency checking of a recursion-free method schema is presented. In Reference [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 Reference [15], a type inference algorithm for a procedural OOPL is proposed. The language presented in the article is polyadic and can express recursion and assignments to local variables. It also provides explicit new and if-then-else expressions. 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 References [14] and [16]). These articles provide type inference algorithms that check type safety for sufficiently practical languages, whereas this paper focuses on what properties of a language make exact type inference possible or impossible.

Our OOPL model is *untyped* in the sense that each variable has no type declaration. In contrast, type-consistency for *typed* OOPLs has 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. Therefore, 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. These articles do not put an assumption to the ability of OOPLs for that reason. Type systems for OOPLs have also been extensively studied [9], [10]. For example, in Reference [10], an elegant type system is proposed that relaxes contravariance restriction. The language presented in the article is polyadic and can express recursion and assignments to local variables, also providing explicit new expressions. In these approaches, an object is basically defined as a record structure, each field of which represents an attribute (a state component) or a method of the object. The main focus is on providing a record structure with a static type system such that (1) the type system reflects inheritance and dynamic method binding, and (2) the type system is safe in the sense that the static type of an object o is always a supertype of the type of o assigned at run-time. These static type systems are defined provided that the signature of each method is statically given, that is, the class to which the returned objects should belong is known in advance. Hence, the computational complexity of type-consistency problem becomes trivial since the analysis of method bodies is unnecessary, as is the case in the consistency problem of typed OOPLs.

Outline of the Thesis

The remainder of this thesis is organized as follows. In Chapter 2, we provide several preliminary definitions such as database schemas, database instances, the type-consistency problem and so on. In Chapter 3, we show the undecidability of the type-consistency problem for retrieval and terminating schemas. In Chapter 4, we discuss the computational complexity of the type-consistency problem for acyclic schemas. Lastly, we summarize this thesis in Chapter 5.

Chapter 2

Preliminary Definitions

2.1 Object-Oriented Database Schemas

2.1.1 Syntax

A *database schema* is a 6-tuple $\mathbf{S} = (C, \preceq, \mathit{Attr}, \mathit{Ad}, \mathit{Meth}, \mathit{Impl})$, where:

1. C is a finite set of *class names.*

2. $\preceq$ is a partial order on C representing a *class hierarchy.* If $c' \preceq c$, 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, i.e., for all c_1, c_2, $c \in C$, either $c_1 \preceq c_2$ or $c_2 \preceq c_1$ whenever $c \preceq c_1$ and $c \preceq c_2$. That is, we do not consider the existence of multiple inheritance. This is not an essential for our results if a proper mechanism is provided for the multiple inheritance.

3. Attr is a finite set of *attribute names.*

4. $\mathit{Ad} : C \times \mathit{Attr} \to C$ is a partial function representing *attribute declarations.* By $\mathit{Ad}(c, a) = c'$, we mean that the value of attribute a of an object of class c must be an object of class c' or its subclasses.

5. Meth is a finite set of *method names.*

6. $\mathit{Impl} : C \times \mathit{Meth} \to \mathit{WFP}$ 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. $y := y'$, 4. $y := m(y')$,
2. $y :=$ self, 5. self.$a := y'$,
3. $y :=$ self.a, 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. A sentence of type 5 is called an *update operation.* 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 three conditions hold:

- No undefined variable is referred to. That is, for each s_i ($i \in [1, n]$), 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 defined attributes are used. That is, for any sentence of type 3, a must be defined at that class or its superclasses. The formal definition of inheritance of attribute declarations will be described later.
- 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 2.1: Consider the three programs in Figure 2.1. 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'). □

Without loss of generality, 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.

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

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

$s_{11}:$ $y :=$ self.a;
$s_{12}:$ self.$a' := y$;
$s_{13}:$ $y' := m(y)$;
$s_{14}:$ return(y').

(a)

$s_{21}:$ $y' :=$ self.a;
$s_{22}:$ self.$a' := y'$;
$s_{23}:$ $y' := m(y)$;
$s_{24}:$ return(y').

(b)

$s_{31}:$ $y :=$ self.a;
$s_{32}:$ self.$a' := y$;
$s_{33}:$ return(y);
$s_{34}:$ $y' := m(y)$.

(c)

Figure 2.1. Example of programs.

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

2.1.2 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 $\preceq$) at which an implementation of m exists, that is, $c' \preceq c''$ for any c'' such that $\mathit{Impl}(c'', m)$ is defined and $c \preceq c''$. 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 $\mathbf{S}$ is a pair $\mathbf{I} = (\nu, \mu)$, where:

1. To each class $c \in C$, ν assigns a disjoint finite set, denoted $\nu(c)$, as *object identifiers* for c. Each $o \in \nu(c)$ is called an object of class c. Let $O_{\mathbf{S},\mathbf{I}} = \bigcup_{c \in C} \nu(c)$. By $\mathit{cl}(o)$, we 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, μ assigns an object, denoted $\mu(o, a)$, which 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'' ($c'' \preceq c'$). Hereafter, we often denote $\mu(o, a)$ by $o.a$.

A *method execution* under a given database instance is formally defined in References [11] and [12]. 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. $y := \mathsf{self}$ means that the self object is assigned to variable y.

3. $y := \mathsf{self}.a$ means that the a-value 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' and the returned value is assigned to y. Let o be the object assigned to y'. If $\mathit{Impl}^*(\mathit{cl}(o), m) = \alpha$, then o is bound to self in α, α is executed, and the returned value is assigned to y. If $\mathit{Impl}^*(\mathit{cl}(o), m)$ is undefined, then a runtime type error occurs.

5. Consider a sentence $\mathsf{self}.a := y'$, and let o be the object assigned to y' when the control reaches this sentence. Assume that the self object is in $\nu(c)$ and $\mathit{Ad}^*(c, a) = c'$. If $\mathit{cl}(o) = c''$ and $c'' \preceq c'$, then the a-value of the self object becomes o. Otherwise, a runtime type error occurs.

A method execution is said to be *terminating* if it terminates successfully or it is aborted due to a runtime type error. Otherwise, it is said to be *nonterminating*. We often say that **S** is *terminating* if every method execution is terminating under every instance of **S**. If **S** is not terminating, then we say that **S** is *nonterminating*.

2.2 The Type-Consistency Problem

Let **S** be a database schema and **I** be a database instance of **S**. We say that **I** is *consistent* under **S** when the following condition holds:

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

Note that if the execution of m on o is nonterminating, then no runtime type error occurs during the execution. If **I** is not consistent under **S**, then we say that **I** is *inconsistent* under **S**. We often say that **S** is *consistent* if there is no inconsistent instance under **S**. If **S** is not consistent, then we say that **S** is *inconsistent*.

The *type-consistency problem* is to decide whether a given database schema is consistent or not.

2.3 Subclasses of Database Schemas

In this section, we define several notions to introduce subclasses of database schemas.

Definition 2.1: The *height* of $\preceq$ in a schema is the maximum integer n such that there exist distinct c_0, c_1, …, $c_n \in C$ satisfying $c_0 \preceq c_1 \preceq \cdots \preceq c_n$. $\square$

If the height of $\preceq$ 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 $\preceq$ is flat.

Definition 2.2: *Impl* is *retrieval* if it includes no update operation (i.e., sentence in the form of "$\mathsf{self}.a := y$"). We often say that **S** is *retrieval* if *Impl* is retrieval. $\square$

Definition 2.3: The *method dependency graph* $G = (V, E)$ of *Impl* is defined as follows [2]:

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

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. $\square$

Definition 2.4: *Ad* is *acyclic* if there exists a partial order $\sqsubseteq$ on C such that $\mathit{Ad}^*(c, a) = c'$ implies $c'' \sqsubseteq c$ and $c'' \neq c$ for all $c'' \preceq c'$. We often say that **S** is *acyclic* if *Ad* is acyclic. $\square$

Suppose that **S** is acyclic. Then, for every $o \in \nu(c)$ and every nonempty sequence $a_1, \ldots, a_n$ of attributes, $o.a_1.\cdots.a_n \notin \nu(c)$. In this case, **S** can be regarded as a nested relational database schema [1], where each class represents a relation.

2.4 Examples

In this section, we provide three examples. The first example shows how to represent Boolean values in update schemas. In the example, a method that calculates NOR and a method that simulates if-then statements are presented. These methods imply the powerful expressiveness of update schemas. The second example shows how to simulate conditional statements in acyclic schemas. Conditional statements can be simulated in acyclic schemas using method name overloading, that is, adopting different method definitions according to the class to which a target object belongs. The third example shows how to simulate iterative statements in acyclic schemas. Iterative statements can be simulated in acyclic schemas using recursive calls and conditional branch described in the second example.

Example 2.2: Let us consider the following schema $\mathbf{S}_{\mathrm{E}} = (C, \preceq, \mathit{Attr}, \mathit{Ad}, \mathit{Meth}, \mathit{Impl})$, where

- $C = \{c, c_t, c_f\}$ such that $c_t \preceq c$ and $c_f \preceq c$ (i.e., c is a superclass of both c_t and c_f, see Figure 2.2(a)); and
- Ad is shown in Figure 2.2(b).

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

Then, we define two methods $\mathsf{nor}[a_1, a_2]$ and $\mathsf{if\_then}[a_1, m]$ as shown in Figure 2.2(c). Method $\mathsf{nor}[a_1, a_2]$ calculates NOR of $o.a_1$ and $o.a_2$, and returns o if the result is true and $o.a_f$ otherwise. Here a' is being used as a form of scratch paper. First, $o.a'$ is initialized with o, i.e., true. Then, $o.a'$ is set $o.a_f$, i.e., false if and only if either $o.a_1$ or $o.a_2$ represents true. Since every Boolean operator can be represented by NORs, we can construct a method which calculates a given Boolean formula using $\mathsf{nor}[a_1, a_2]$. On the other hand, $\mathsf{if\_then}[a_1, m]$ simulates if-then statements: m is invoked on o if and only if $o.a_1 = o$. By the first two lines of $(c_t, \mathsf{if\_then}[a_1, m])$, $o.a''$ is "normalized" so that $o.a'' = o.a_f$ (and hence $cl(o.a'') \neq c_t$) whenever $o.a_1$ represents false. □

$$\begin{tikzcd}
 & c & \\
c_t \arrow[ur] & & c_f \arrow[ul]
\end{tikzcd}$$

(a)

Class c_t		
a_1, a_2, a', a''	:	c
a_f	:	c_f

(b)

```
$(c_t, \mathsf{nor}[a_1, a_2])$ :
$\mathsf{self}.a' := \mathsf{self}$;
$y := \mathsf{nor}'(\mathsf{self}.a_1)$;
$y := \mathsf{nor}'(\mathsf{self}.a_2)$;
$\mathsf{return}(\mathsf{self}.a')$.
```

```
$(c_t, \mathsf{nor}')$ :
$\mathsf{self}.a' := \mathsf{self}.a_f$;
$\mathsf{return}(\mathsf{self})$.
```

```
$(c, \mathsf{nor}')$ :
$\mathsf{return}(\mathsf{self})$.
```

```
$(c_t, \mathsf{if\_then}[a_1, m])$ :
$\mathsf{self}.a'' := \mathsf{nor}[a_1, a_1](\mathsf{self})$;
$\mathsf{self}.a'' := \mathsf{nor}[a'', a''](\mathsf{self})$;
$y := \mathsf{if\_then}'[m](\mathsf{self}.a'')$;
$\mathsf{return}(y)$.
```

```
$(c_t, \mathsf{if\_then}'[m])$ :
$y := m(\mathsf{self})$;
$\mathsf{return}(y)$.
```

```
$(c, \mathsf{if\_then}'[m])$ :
$\mathsf{return}(\mathsf{self})$.
```

(c)

Figure 2.2. Definition of $\mathbf{S}_{\mathrm{E}}$.

```
(c, if_then) :
y := if_then(self.a);
return(self).
```

```
(c_t, if_then) :
y := action(self);
return(self).
```

```
(c_f, if_then) :
return(self).
```

Figure 2.3. Definition of *Impl* of $\mathbf{S}_{\mathrm{I}}$.

Example 2.3: Let us consider the following schema $\mathbf{S}_{\mathrm{I}} = (C, \preceq, \mathit{Attr}, \mathit{Ad}, \mathit{Meth}, \mathit{Impl})$, where

- $C = \{c, c_t, c_f\}$;
- $c_f \preceq c_t$;
- $\mathit{Attr} = \{a\}$;
- $\mathit{Ad}(c, a) = c_t$, and both c_t and c_f have no attribute declarations;
- $\mathit{Meth} = \{\mathsf{if\_then}, \mathsf{action}\}$; and
- *Impl* is shown in Figure 2.3, where the definition of action is omitted. Any method can be action as long as it is well defined.

Here, class c_t represents true, and class c_f represents false. Note that the Boolean-value representation used here is different from that in Example 2.2. Consider the execution of method if_then on $o \in \nu(c)$. Then, it is easy to see that method action is invoked on $o.a$ if and only if $o.a$ is in $\nu(c_t)$, that is, $o.a$ represents true. In this way, conditional branch can be expressed by utilizing the class to which a target object belongs. □

Example 2.4: Let us consider the following schema $\mathbf{S}_{\mathrm{W}} = (C, \preceq, \mathit{Attr}, \mathit{Ad}, \mathit{Meth}, \mathit{Impl})$, where for each $i \in [0, n-1]$

```
(c_{i,t}, while) :
y := check_con(self.a);
return(self).
```

```
(c_{n,t}, while) :
return(self).
```

for each $i \in [1, n]$

```
(c_{i,t}, check_con) :
y := body(self);
y := while(self);
return(self).
```

```
(c_{i,f}, check_con) :
return(self).
```

Figure 2.4. Definition of *Impl* of $\mathbf{S}_{\mathrm{W}}$.

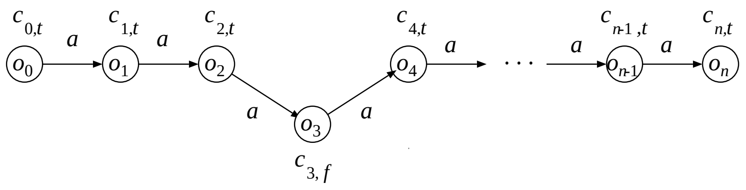


Figure 2.5. Example of a database instance of $\mathbf{S}_{\mathrm{W}}$.

- $C = \{c_{0,t}, c_{1,t}, \ldots, c_{n,t}, c_{1,f}, \ldots, c_{n,f}\}$, where n is a positive integer;
- $c_{i,f} \preceq c_{i,t}$ for each $i \in [1, n]$;
- $Attr = \{a\}$;
- $Ad(c_{i,t}, a) = c_{i+1,t}$ for each $i \in [0, n-1]$;
- $Meth = \{\mathsf{while}, \mathsf{check\_con}, \mathsf{body}\}$; and
- *Impl* is shown in Figure 2.4. Also, the definition of body is omitted.

An example of a database instance of $\mathbf{S}_{\mathrm{W}}$ is shown in Figure 2.5. Iterative statements can be expressed by the combination of recursive calls and conditional branch. Consider the execution of method while on $o_0 \in \nu(c_{0,t})$. As stated before, if $o_0.a = o_1$ is in $\nu(c_{1,t})$, that is, if $o_0.a$ represents true, then method body is performed. After that, method while is invoked recursively on o_1. When the a-value of a target object of method while represents false or the "tail" object becomes a target object, the method execution terminates. In the above example, the execution terminates when while is invoked on o_2.

In summary, method while, method check_con, and method body each can be considered as a while statement, a condition of the while statement, and a body of the while statement, respectively. □

Chapter 3

Undecidability Results of the Type-Consistency Problem

3.1 Retrieval Schemas

Theorem 3.1: The type-consistency problem for retrieval schemas is undecidable, even if the height of the class hierarchy is one. $\Box$

This theorem is proved by showing a reduction from the Post's Correspondence Problem (PCP) to the consistency problem for a database schema with the conditions 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 the PCP over alphabet $\Sigma = \{0, 1\}$. We construct a database schema $\mathbf{S}_{w,u} = (C, \preceq, \mathit{Attr}, \mathit{Ad}, \mathit{Meth}, \mathit{Impl})$ such that

- $\mathbf{S}_{w,u}$ is retrieval;
- the height of $\preceq$ of $\mathbf{S}_{w,u}$ is one; 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 $o_1 \in O_{\mathbf{S}_{w,u},\mathbf{I}}$ 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

$$\begin{tikzcd}
 & & c & & & c' & \\
c_1 \arrow[urr] & c_2 \arrow[ur] & \bullet\ \bullet\ \bullet & c_n \arrow[ul] & c'_0 \arrow[ur] & & c'_1 \arrow[ul]
\end{tikzcd}$$

Figure 3.1. Definition of $\preceq$ of $\mathbf{S}_{w,u}$.

for each $i \in [1, n]$

Class c_i	
$a_{\rightarrow}$:	c

Class c	
$a_{\Rightarrow}$:	c'

Class c'_0, c'_1	
$a_{\Rightarrow}$:	c'

Figure 3.2. Definition of Ad of $\mathbf{S}_{w,u}$.

o_1 under $\mathbf{I}$ is aborted. Otherwise, the execution of post for o_1 under $\mathbf{I}$ is non-terminating (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 Σ.

Figures 3.1 and 3.2 show the definitions of $\preceq$ and Ad of $\mathbf{S}_{w,u}$, respectively. Class c_i ($i \in [1, n]$) represents the i-th pair $\langle w_i, u_i \rangle$, and class c'_0 (resp. c'_1) represents symbol 0 (resp. 1). Next, define methods post, mw, is0, is1, and isc$'$ as Figures 3.3–3.6 (also define method mu similarly to mw). The underlined part (e.g., the second line of (c_i, mw)) 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

```
for each $i \in [1, n]$
($c_i$, post)
$y$ := mw(self);
$y$ := isc$'$($y$);
$y$ := mu(self);
$y$ := isc$'$($y$);
$y$ := test(self);
return(self).
```

Figure 3.3. Definition of method post.

- method test, which appears at the fifth line of (c_i, 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, 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, post) during the execution of post for some $o_1 \in O_{\mathbf{S}_{w,u},\mathbf{I}}$ 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 Figure 3.7):

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

In $\mathbf{I}$, 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 Σ. Let w_{o_i} and u_{o_i} denote the words represented by o_i (i.e., $w_{o_i} = w_h$ and $u_{o_i} = u_h$ if $o_i \in \nu(c_h)$), 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

```
for each $i \in [1, n]$
$(c_i, \mathsf{mw})$ :
$y := \mathsf{mw}(\mathsf{self}.a_{\rightarrow})$;
if $w_{i,d_i}$ is 0
    then $y := \mathsf{is0}(y)$;
    else $y := \mathsf{is1}(y)$;
        ⋮
if $w_{i,1}$ is 0
    then $y := \mathsf{is0}(y)$;
    else $y := \mathsf{is1}(y)$;
return($y$).
```

```
$(c, \mathsf{mw})$ :
return($\mathsf{self}.a_{\Rightarrow}$).
```

```
$(c', \mathsf{mw})$ :
return(self).
```

Figure 3.4. Definition of method mw.

lemmas claim that the execution of the first two lines of $(cl(o_1), \mathsf{post})$ terminates if and only if $w_{o_1} \cdots w_{o_k} = x_l \cdots x_1$.

Lemma 3.1: Suppose that $\mathbf{I}$ 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, mw), method mw is recursively invoked on $o_1.a_{\rightarrow}$, which is an object of class c since $k = 1$. By (c, mw), this invocation results in o'_1, and it is assigned to y at the first line of (c_1, mw). Suppose that $w_{1,d_1} = 0$. By the second line of (c_1, mw), 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, mw) terminates and o'_2 is assigned to y if and only if $w_{1,d_1} = x_1$. By induction on d_1, we can show that the execution of mw 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.

$(c'_0, \mathsf{is0})$:
$\mathsf{return}(\mathsf{self}.a_{\Rightarrow})$.

$(c'_1, \mathsf{is1})$:
$\mathsf{return}(\mathsf{self}.a_{\Rightarrow})$.

$(c', \mathsf{is0})$:
$\underline{\text{loop forever}}$.

$(c', \mathsf{is1})$:
$\underline{\text{loop forever}}$.

$(c, \mathsf{is0})$:
$\mathsf{return}(\mathsf{self})$.

$(c, \mathsf{is1})$:
$\mathsf{return}(\mathsf{self})$.

Figure 3.5. Definitions of methods is0 and is1.

(c'_0, isc') :
$\underline{\text{loop forever}}$.

(c'_1, isc') :
$\underline{\text{loop forever}}$.

(c', isc') :
$\mathsf{return}(\mathsf{self})$.

(c, isc') :
$\mathsf{return}(\mathsf{self})$.

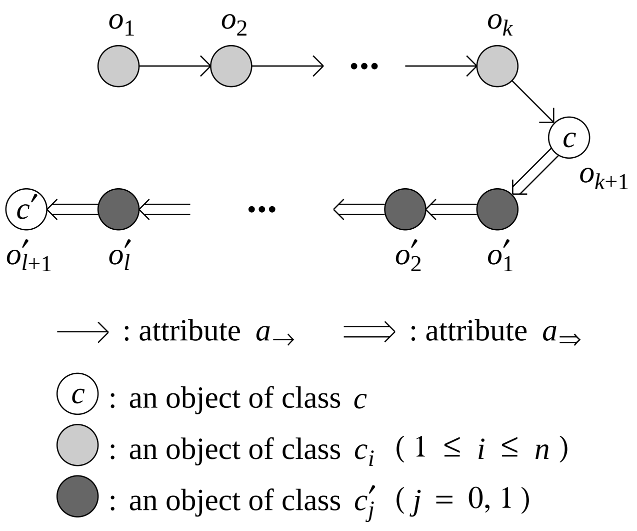
Figure 3.6. Definition of method isc$'$.

[Inductive Step] Suppose that $k > 1$. By the first line of (c_1, mw), method mw is recursively invoked on $o_1.a_{\rightarrow}$ $(= o_2)$. From the inductive hypothesis, the execution of mw 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, mw), 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 3.2: Suppose that $\mathbf{I}$ 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 $w_{o_1} \cdots w_{o_k} = x_l \cdots x_1$. Similar properties 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 $w_{o_1} \cdots w_{o_k} = u_{o_1} \cdots u_{o_k} = x_l \cdots x_1$.



o_1
o_2
o_k
c
o_{k+1}
c'
o'_{l+1}
o'_l
o'_2
o'_1
: attribute $a_\rightarrow$
: attribute $a_\Rightarrow$
: an object of class c
: an object of class c_i ($1 \leq i \leq n$)
: an object of class c'_j ($j = 0, 1$)

Figure 3.7. Database instance of $\mathbf{S}_{w,u}$.

Next, suppose that $\mathbf{I}$ is not in the form of (F1). Then, $\mathbf{I}$ 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 mw at the first line of (c_i, mw) does not terminate. In the case of (F3), the execution of is0 or is1 in (c_i, mw), or isc$'$ in $(cl(o_1), \mathsf{post})$ does not terminate. Therefore, if $\mathbf{I}$ 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 $\mathbf{I}$ 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 $\mathbf{I}$, the control reaches the fifth line of $(cl(o_1), \mathsf{post})$. Conversely, suppose that there is an instance $\mathbf{I}$ such that the control reaches the fifth line of $(cl(o_1), \mathsf{post})$ during the execution of post for o_1 under $\mathbf{I}$. Then, $\mathbf{I}$ 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 Theorem 3.1.

As stated in Chapter 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,

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

where $\mathsf{isX}_{i,j}$ is either $\mathsf{is0}$ or $\mathsf{is1}$ according to $w_{i,j}$, and $\mathsf{ma}_{\rightarrow}$ 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 3.1: The type-consistency problem for method schemas with methods of arity at most two is undecidable. $\Box$

3.2 Terminating Schemas

Theorem 3.2: The type-consistency for terminating schemas is undecidable, even if the height of the class hierarchy is one. $\Box$

To prove Theorem 3.2, for a given input string x of a fixed deterministic Turing machine M, we construct a schema $\mathbf{S}_{M,x} = (C, \preceq, \mathit{Attr}, \mathit{Ad}, \mathit{Meth}, \mathit{Impl})$ satisfying the following conditions:

- $\mathbf{S}_{M,x}$ is terminating;
- the height of $\preceq$ of $\mathbf{S}_{M,x}$ is one; 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 3.1: A *deterministic Turing machine* M is a triple (Q, Σ, δ), where

- Q is a finite set of states. Q contains two special states: the initial state q_0 and the accepting state q_{yes};
- Σ is a finite set of symbols. Σ contains two special symbols: the *blank symbol* B and the *first symbol* $\rhd$. The first symbol is always placed at the leftmost cell of the tape;
- δ is a function which maps $(Q - \{q_{\text{yes}}\}) \times \Sigma$ to $Q \times \Sigma \times \{\leftarrow, \rightarrow, -\}$. We assume that if $\delta(q, \rhd) = (q', \gamma, d)$, then $\gamma = \rhd$ 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 $\gamma_1 = \rhd$, 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 $\vdash_{\overline{M}}$ over the set of IDs is defined as usual. $\square$

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_{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 2.2), then m sets $o.a_{\text{ws}}$ false and continues 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_{ws}. See Lemma A.1 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 (≥ 0) is a constant dependent on $\mathbf{I}$. If the ID after r-step transitions contains the accepting state q_{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 by $\mathbf{I}$ are large enough to

$$\begin{tikzcd}
& c & \\
c_{\mathrm{t}} \arrow[ur] & c'_{\mathrm{t}} \arrow[u] & c_{\mathrm{f}} \arrow[ul]
\end{tikzcd}$$

Figure 3.8. Definition of $\preceq$ of $\mathbf{S}_{M,x}$.

Class c_{t}		
$a_{\Rightarrow}$	:	c
$\vec{a}, \vec{a}', \vec{a}''$	:	c
a_{cont}	:	c
$a_{\mathrm{yes}}, a'_{\mathrm{yes}}$	:	c
a_{f}	:	c_{f}
$a_{\mathrm{ws}}, a'_{\mathrm{ws}}$	:	c

Class c'_{t}		
a'_{yes}	:	c
a_{f}	:	c_{f}

Class c		
a'_{t}	:	c'_{t}

Figure 3.9. Definition of Ad of $\mathbf{S}_{M,x}$.

find an aborted execution of TM under $\mathbf{I}$ (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 $\preceq$ and Ad of $\mathbf{S}_{M,x}$ as shown in Figures 3.8 and 3.9, respectively. In Figure 3.9, $\vec{a}$ 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 $i \in [1, K]$). An element of an ID is stored in $\vec{a}$ as the binary encoded form stated in Example 2.2. Attributes $\vec{a}'$ and $\vec{a}''$ are used for storing intermediate results during the computation of an ID. Attribute a_{cont} is used for determining r, i.e., the number of steps to be simulated. Attributes a_{yes} and a'_{yes} are used for checking whether M is in the accepting state or not after the simulation. Note that the height of $\preceq$ is one. Next, define method TM as shown in Figure 3.10. All the methods except test is defined at every class. Method test is defined only at class c_{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

```
$(c_{\mathrm{t}}, \mathsf{TM})$ :
$y$ := get_ws(self);
$y$ := initws(self);
$y$ := step(self);
$y$ := accept(self);
$y$ := test($y$);
return(self).
```

Figure 3.10. Definition of method TM.

of $(c_{\mathrm{t}}, \mathsf{TM})$ and test is about to be invoked on an object of class c, c_{t}, or c'_{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 $(c_{\mathrm{t}}, \mathsf{TM})$. This obtains objects $o_2, \ldots, o_{k+1}$ satisfying $o_i.a_{\Rightarrow} = o_{i+1}$ $(i \in [1,k])$ by following attribute $a_{\Rightarrow}$ of each o_i, where k is a constant dependent on $\mathbf{I}$ 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_{t}, the class of $o_2, \ldots, o_k$ must be c_{t}. By a technical reason, we want o_{k+1} to be an object of class c'_{t}. To achieve this, if the $a_{\Rightarrow}$-chain from o_1 (1) ends up with an object of class c_{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'_{t} (see Figure 3.11). Lemma A.3 in Appendix A provides a formal description of the behavior of get_ws.

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 $(c_{\mathrm{t}}, \mathsf{TM})$, each $I_0[i]$ $(i \in [1,k])$ 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 $\langle \bot, B \rangle$ 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 A.4 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, $I_j[i]$ $(i \in [1, k-j])$ is stored in

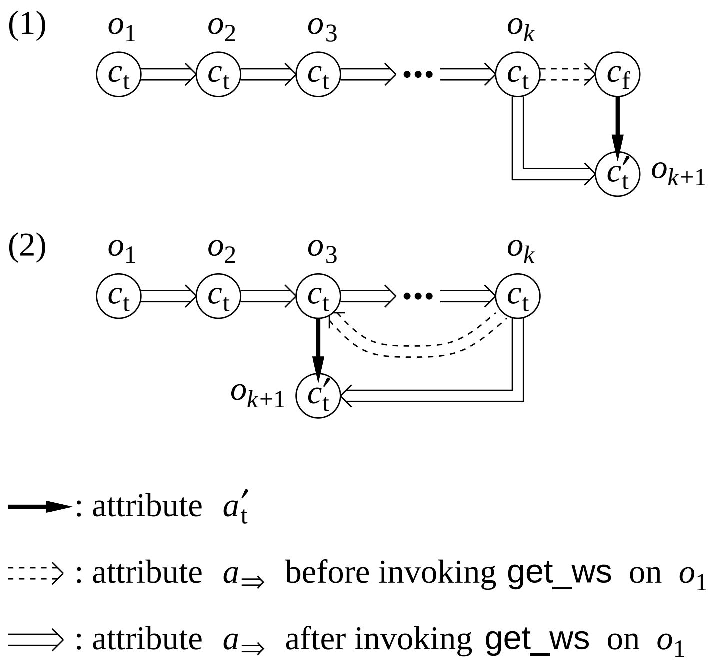

(1)
o_1
o_2
o_3
o_k
c_t
c_f
c_t'
o_{k+1}
(2)
o_1
o_2
o_3
o_k
c_t
c_t'
o_{k+1}
: attribute a_t'
: attribute $a_\Rightarrow$ before invoking get_ws on o_1
: attribute $a_\Rightarrow$ after invoking get_ws on o_1

Figure 3.11. Database instance after invoking method get_ws on o_1.

$o_{j+i}.\vec{a}$. Note that the initial ID I_0 satisfies this condition. Consider a database instance shown in Figure 3.12(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 2.2. 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 Figure 3.12(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 Figure 3.13 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 Figure 3.12(c), where $\delta(q,1) = (q', 0, \rightarrow)$). Next, we explain attribute a_{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 $o.a_{\mathrm{cont}}$ represents true, then the simulation of M is continued. Otherwise, the simulation stops. For example, in the case of Figure 3.12(c), the simulation stops

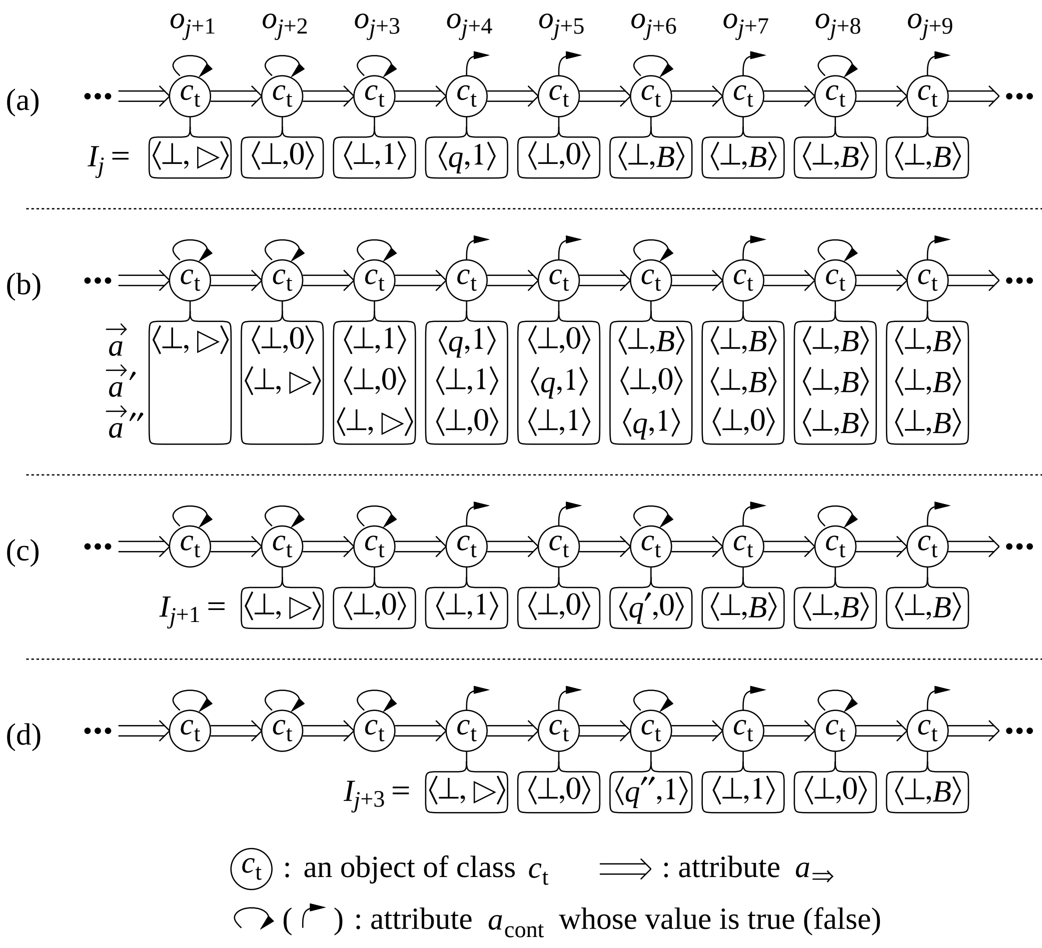

o_{j+1} o_{j+2} o_{j+3} o_{j+4} o_{j+5} o_{j+6} o_{j+7} o_{j+8} o_{j+9}
(a)
$I_j = \langle\perp,\triangleright\rangle\ \langle\perp,0\rangle\ \langle\perp,1\rangle\ \langle q,1\rangle\ \langle\perp,0\rangle\ \langle\perp,B\rangle\ \langle\perp,B\rangle\ \langle\perp,B\rangle\ \langle\perp,B\rangle$
(b)
$\vec{a}$ $\vec{a}'$ $\vec{a}''$
(c)
$I_{j+1} = \langle\perp,\triangleright\rangle\ \langle\perp,0\rangle\ \langle\perp,1\rangle\ \langle\perp,0\rangle\ \langle q',0\rangle\ \langle\perp,B\rangle\ \langle\perp,B\rangle\ \langle\perp,B\rangle$
(d)
$I_{j+3} = \langle\perp,\triangleright\rangle\ \langle\perp,0\rangle\ \langle q'',1\rangle\ \langle\perp,1\rangle\ \langle\perp,0\rangle\ \langle\perp,B\rangle$
c_t : an object of class c_t
$\Longrightarrow$: attribute $a_\Rightarrow$
( ) : attribute a_{cont} whose value is true (false)

Figure 3.12. Working space to simulate M.

```
$(c_{\mathrm{t}}, \mathsf{copy}[a_1, a_2])$ :
$y :=$ set_f$[a_2]$(self.$a_{\Rightarrow}$);
$y :=$ if_then$[a_1,$ set_t$[a_2]]$(self);
return(self).
```

```
$(c_{\mathrm{t}}, \mathsf{set\_t}[a_2])$ :
$y :=$ set_t$'[a_2]$(self.$a_{\Rightarrow}$);
return(self).
```

```
$(c_{\mathrm{t}}, \mathsf{set\_f}[a_2])$ :
self.$a_2 :=$ self.$a_{\mathrm{f}}$;
return(self).
```

```
$(c_{\mathrm{t}}, \mathsf{set\_t'}[a_2])$ :
self.$a_2 :=$ self;
return(self).
```

```
$(c, \mathsf{set\_f}[a_2])$ :
return(self).
```

```
$(c, \mathsf{set\_t'}[a_2])$ :
return(self).
```

Figure 3.13. Definition of method copy$[a_1, a_2]$.

after two steps (Figure 3.12(d)). See Lemmas A.5 and A.6 in Appendix A for a formal description of the behavior of step.

Method accept checks whether q_{yes} is in the last ID by using nor$[*, *]$ and copy$[*, *]$. It returns $o_{k+1} \in \nu(c'_{\mathrm{t}})$ if q_{yes} is in the last ID, and $o_{k+1}.a_{\mathrm{f}} \in \nu(c_{\mathrm{f}})$ otherwise. See Lemma A.7 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_{f}, this invocation causes a type error if and only if q_{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_{yes} never appears in the $a_{\Rightarrow}$-chain, invocation of test causes no type error. Thus, Theorem 3.2 has been proved.

Chapter 4

Complexity of the Type-Consistency Problem for Acyclic Schemas

4.1 The Upper Bound for Acyclic Schemas

In this section, we show that the type-consistency problem for acyclic schemas is in coNEXPTIME.

Lemma 4.1: Let $\mathbf{S}$ be an acyclic schema. For any occurrence of any variable y in any method definition, the value of y is

- always a self object under any instance of $\mathbf{S}$; or
- always a *non*-self object under any instance of $\mathbf{S}$

in the occurrence.

Proof: Assume that there is an occurrence of a variable in $\mathit{Impl}^*(c, m)$ such that the variable can take both of the values in the occurrence, a self object and a non-self object, depending on database instances. Let s_1 be the first sentence in $\mathit{Impl}^*(c, m)$ among the sentences which contain such occurrences. If s_1 had the form "$y := y'$", then another sentence which assigns objects to y' would be the first one which contains such occurrences. If s_1 had the form "$y := \mathsf{self}$" or

"$y := \mathsf{self}.a$", then the value of y would always be a self object or always be a non-self object, respectively. Therefore, s_1 must have the form "$y := m'(y')$". If the value of y' is always a non-self object, then the returned value cannot be a self object because it is not reachable from a non-self object when Ad is acyclic. Therefore, the value of y' must be always a self object. If $m' = m$, then the execution of m falls into a loop, and no object is assigned to y. Thus, $m' \neq m$. Let $\mathsf{return}(y'')$ be the last sentence in $\mathit{Impl}^*(c, m')$. Then, the value of y'' can be both of a self object and a non-self object, so there is an occurrence of a variable such that the variable can take both of the values, a self object and a non-self object. Let s_2 be the first sentence in $\mathit{Impl}^*(c, m')$ among the sentences which contain such occurrences. By the same discussion, s_2 must have the form "$y := m''(y')$" and $m'' \neq m'$, $m'' \neq m$. By repeating this argument, we have a contradiction of the finiteness of method names. □

Definition 4.1: Let **S** be an acyclic schema. A method m of **S** is said to be *strongly recursive* w.r.t. c if there is a sequence $m_1, m_2, \ldots, m_k, m_{k+1} = m_1$ such that $\mathit{Impl}^*(c, m_i)$ includes "$y' := m_{i+1}(y)$" and the value of y is always a self object for any $i \in [1, k]$. □

From the acyclicity of **S** and Lemma 4.1, the following two lemmas are derived.

Lemma 4.2: Let m be a method of an acyclic schema. If the execution of $m(o)$ does not terminate, then a strongly recursive method must be invoked during the execution of $m(o)$.

Proof: Suppose that the execution of $m(o)$ does not terminate. Since every method definition contains only a finite number of sentences, there must be infinite method invocations during the execution of $m(o)$. Since the numbers of methods and objects are both finite, there exist m' and o' such that m' is invoked on o' infinitely often during the execution of $m(o)$. From the acyclicity of the schema, all method invocations during the execution $m'(o')$ are on o', i.e., the self object of method m'. (Otherwise, o' could not be a target object of m' after the first invocation of m' on o'.) Therefore, from Definition 4.1, m' is a strongly recursive method. □

Lemma 4.3: The execution of a strongly recursive method never terminates unless it causes a type error.

Proof: Let m_1,. . . , m_k be a sequence of methods which satisfies the condition of Definition 4.1. Consider the execution of $m_1(o)$, where m_1 is strongly recursive. First, assume that the execution does not reach the sentence $y' := m_2(y)$. In this case, either a type error has occurred before the sentence or the execution has already been in an infinite loop. Then, assume that the execution reaches the sentence $y' := m_2(y)$. In this case, from Definition 4.1, m_2 must be invoked on o. By applying the same discussion to m_2 and all the subsequent methods repeatedly, it can be shown that m_1 is invoked on o recursively unless a type error has occurred during the execution or the execution has already in an infinite loop. Again, the same discussion can be applied to m_1, and thus the lemma holds. $\square$

Theorem 4.1: The type-consistency problem for acyclic schemas is in coNEXPTIME.

Proof: A database instance of an acyclic schema is represented as a directed acyclic graph (dag), where nodes are objects and edges are attribute-value relationships between objects. The maximum length of a path of a dag is bounded by $|C|$ because of the acyclicity of a schema. And each node has at most $|Attr|$ edges. Thus, the total number of objects which are reachable from an object, that is, the total number of nodes in a dag, is bounded by $|Attr|^{|C|}$. So it suffices to guess method m, object o, and instance $\mathbf{I}$ whose size is at most $|Attr|^{|C|}$, and then examine whether the execution of $m(o)$ under $\mathbf{I}$ causes a type error. However, to prevent the simulation of the execution from falling into a loop, we stop the simulation and decide the guess failed when m' is invoked on o' during the execution of $m'(o')$ (that is, m' is found to be strongly recursive w.r.t. $cl(o')$). Thus, from Lemma 4.2, the simulation always terminates. Furthermore, it is easily shown that the simulation takes at most exponential time of $||\mathbf{S}||$.

In the following, we show that we never miss an instance which causes a type error even if we stop the simulation. From Lemma 4.3, sentences after an invocation of a strongly recursive method are never executed, and therefore it suffices to check whether the invocation causes a type error. Assume that m' is strongly recursive w.r.t. c, and there exists an instance $\mathbf{I}$ such that a type error occurs during the n-th execution of m' on $o' \in \nu(c)$. Then, a type error can be detected when we guess m', o', and $\mathbf{I}'$ which is an instance immediately before the n-th invocation of m' on o' under $\mathbf{I}$. $\square$

	0	1	2	3
0	t_0	t_1	t_3	t_2
1	t_2	t_1	t_0	t_1
2	t_1	t_3	t_2	t_1
3	t_1	t_0	t_1	t_3

Figure 4.1. Example of a 4×4 tiling.

4.2 The Lower Bound for Recursion-Free Acyclic Schemas

In this section, we show that the type-consistency problem for recursion-free acyclic schemas is coNEXPTIME-hard by reducing the complement of the TILING problem, which is known to be NEXPTIME-compelete [13].

4.2.1 Overview of the Reduction

The TILING problem is defined as follows [13].

Definition 4.2: We are given a set of square tile types $T = \{t_0, \ldots, t_{k-1}\}$, together with two relations $H, V \subseteq T \times T$ (the horizontal and vertical compatibility relations, respectively). We are also given an integer n in binary. An $n \times n$ *tiling* is a function $f : \{0, \ldots, n-1\} \times \{0, \ldots, n-1\} \rightarrow T$ such that (t-a) $f(0,0) = t_0$, (t-b) for all i, j, $(f(i,j), f(i,j+1)) \in H$, and (t-c) for all i, j, $(f(i,j), f(i+1,j)) \in V$. *TILING* is the problem of deciding, given T, H, V, and n, whether an $n \times n$ tiling exists. □

Example 4.1: Assume that we are given

$$\begin{aligned} T &= \{t_0, t_1, t_2, t_3\}, \\ H &= \{(t_0, t_1), (t_1, t_0), (t_1, t_3), (t_2, t_1), (t_3, t_2)\}, \\ V &= \{(t_0, t_2), (t_1, t_1), (t_1, t_3), (t_2, t_1), (t_3, t_0)\}, \end{aligned}$$

and $n = 100_2$ $(= 4)$. Then, the function shown in Figure 4.1 is a 4×4 tiling. For example, $(f(0,0), f(0,1)) = (t_0, t_1) \in H$, $(f(0,0), f(1,0)) = (t_0, t_2) \in V$ and so on. $\square$

Theorem 4.2: The type-consistency problem for recursion-free acyclic schemas is coNEXPTIME-hard even if the height of the class hierarchy is at most one. $\square$

We construct a recursion-free acyclic schema $\mathbf{S}_\mathrm{T}$ such that $\mathbf{S}_\mathrm{T}$ is inconsistent if and only if an $n \times n$ tiling exists. We can assume $n = 2^m$ for some integer m because it can be shown that TILING is NEXPTIME-complete even when n is a power of 2. Figure 4.2 shows an "ideal" instance of $\mathbf{S}_\mathrm{T}$, where $n = 4$ and $k = 4$. The meaning of "ideal" is formally defined later (see Section 4.2.2), but its intuitive meaning is that we can construct methods such that a type error occurs on an ideal instance if and only if $n \times n$ tiling exists. As we shall see, if an instance is not ideal, then no type error occurs. The upper half of an ideal instance represents a candidate for an $n \times n$ tiling, while the lower half is used for checking the compatibility given by H and V. The result of the compatibility check is stored as the a-value of the "tail" object o_f. Generally, $\mathbf{S}_\mathrm{T} = (C, \preceq, \mathit{Attr}, \mathit{Ad}, \mathit{Meth}, \mathit{Impl})$ is constructed as follows:

- Definitions of C and $\preceq$ are shown in Figure 4.3; and
- Definition of Ad is shown in Figure 4.4. Note that Ad is acyclic.

Each of classes $c^t_0, \ldots, c^t_{k-1}$ corresponds to a type of tiles, $t_1, \ldots, t_{k-1}$, respectively.

Classes which have subscripts t and f are used for representing the boolean values, true and false, respectively. For example, let us consider classes $c_{\mathrm{com,t}}$ and $c_{\mathrm{com,f}}$. Let o be an object in $\nu(c'_0)$. Then, by setting the a-value of o to $o.a_t$ or $o.a_f$ depending on the situation, we can store one boolean value as the a-value of o. That is, we can interpret that $o.a$ represents true if $o.a$ is in $\nu(c_{\mathrm{com,t}})$, and false if $o.a$ is in $\nu(c_{\mathrm{com,f}})$.

Classes $c_{3,e}, \ldots, c_{2m,e}$ are used for holding intermediate results of checking whether an instance is ideal.

Classes that are not specified in Figure 4.4 have no explicit attribute declarations. That is, classes $c_{3,e}, \ldots, c_{2m,e}$, c'_{fin}, $c'_{\mathrm{fin,t}}$ and $c'_{\mathrm{fin,f}}$ have no attributes, and the

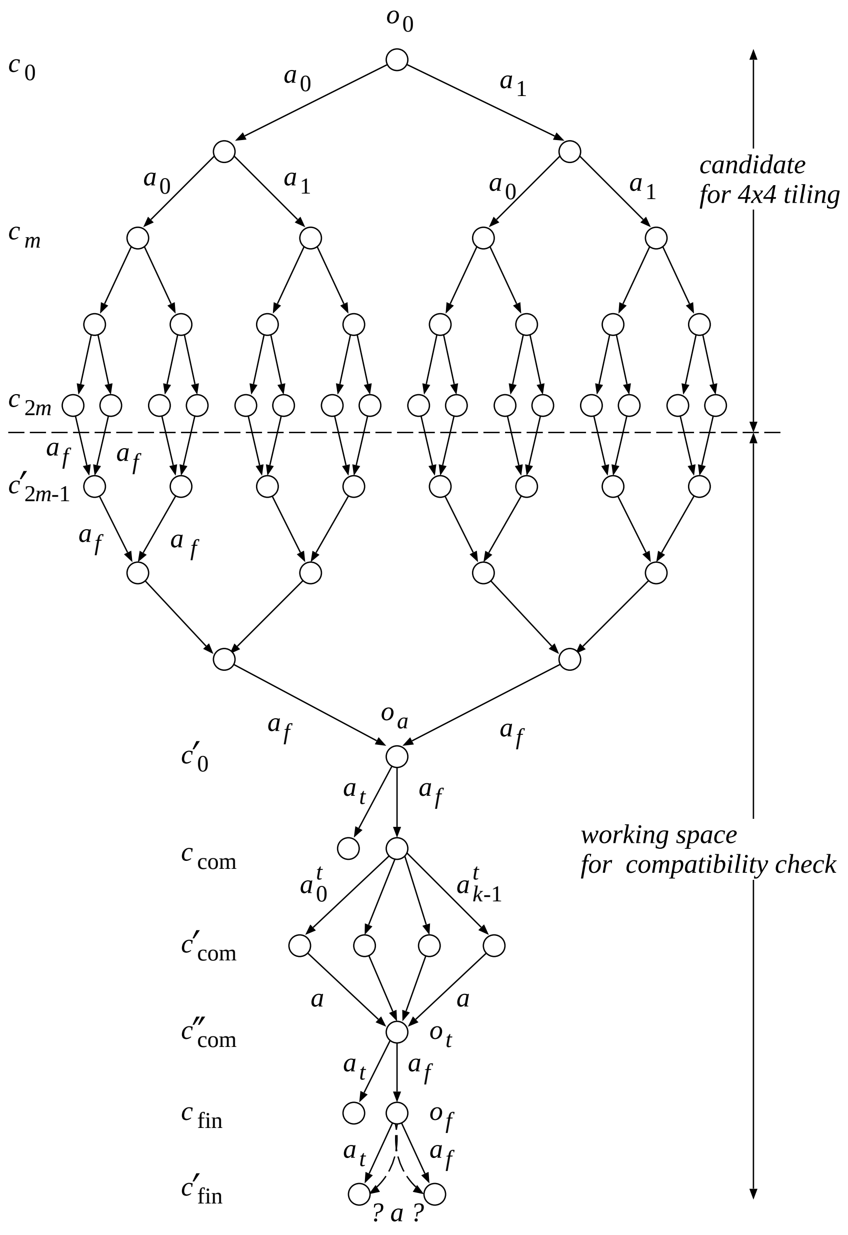


Figure 4.2. Ideal database instance of $\mathbf{S}_{\mathrm{T}}$.

$$\begin{aligned} C \ = \{ \ & c_0, \ldots, c_{2m}, c_{3,e}, \ldots, c_{2m,e}, \\ & c'_0, \ldots, c'_{2m-1}, c'_{0,t}, \ldots, c'_{2m-1,t}, c'_{0,f}, \ldots, c'_{2m-1,f}, \\ & c_{\text{com}}, c_{\text{com,t}}, c_{\text{com,f}}, c'_{\text{com}}, c''_{\text{com}}, c_{\text{fin}}, c_{\text{fin,t}}, c_{\text{fin,f}}, \\ & c'_{\text{fin}}, c'_{\text{fin,t}}, c'_{\text{fin,f}}, c^t_0, \ldots, c^t_{k-1}, c^{t'}_0, \ldots, c^{t'}_{k-1}\}; \end{aligned}$$

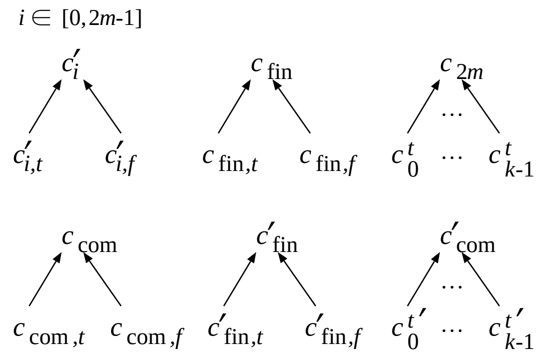


Figure 4.3. Definitions of C and $\preceq$ of $\mathbf{S}_\mathrm{T}$.

other unspecified classes such as $c_{\text{com,t}}$, $c_{\text{com,f}}$, $c_{\text{fin,t}}$ and $c_{\text{fin,f}}$ inherit the attribute declarations from their parent classes and have no extra attributes.

- Definition of method tiling is shown in Figure 4.5. Other method definitions in *Impl* are omitted.

Intuitively, method tiling checks whether a candidate for an $n \times n$ tiling which is represented by an instance of $\mathbf{S}_\mathrm{T}$ is actually an $n \times n$ tiling. As we have described above, an instance needs to be ideal in order to be able to check this condition properly. First, method icheck checks whether an instance is ideal. Next, method tcheck, which is invoked only when the instance is ideal, checks whether the candidate is an $n \times n$ tiling. If the instance is ideal and the candidate is an $n \times n$ tiling, then a type error occurs at the invocation of method test.

Let $\mathbf{I} = (\nu, \mu)$ be a (possibly non-ideal) database instance of $\mathbf{S}_\mathrm{T}$. The candidate function $f_\mathbf{I}$ is defined as follows: Let $i_0 \cdots i_{m-1}$ be the binary representation for each $i \in [0, 2m-1]$

class c_i		
a_0, a_1	:	c_{i+1}

for each $i \in [0, m-1]$
and $j \in [i+1, i+m]$

class c_j		
a_0^i, a_1^i	:	c_{j+1}

for each $i \in [2, 2m-1]$

class c_j		
a	:	c_{i+1}
a_e	:	$c_{i+1,e}$

class c_{2m}		
a	:	c'_{2m-1}
a_t	:	$c'_{2m-1,t}$
a_f	:	$c'_{2m-1,f}$

for each $i \in [1, 2m-1]$

class c'_i		
a	:	c'_{i-1}
a_t	:	$c'_{i-1,t}$
a_f	:	$c'_{i-1,f}$

class c'_0		
a	:	c_{com}
a_t	:	$c_{\text{com,t}}$
a_f	:	$c_{\text{com,f}}$

class c_{com}		
a_i^t	:	$c_i^{t'}$ $(i \in [0, k-1])$
a	:	c'_{com}

class c'_{com}		
a	:	c''_{com}

class c''_{com}		
a	:	c_{fin}
a_t	:	$c_{\text{fin,t}}$
a_f	:	$c_{\text{fin,f}}$

class c_{fin}		
a	:	c'_{fin}
a_t	:	$c'_{\text{fin,t}}$
a_f	:	$c'_{\text{fin,f}}$

Figure 4.4. Definition of Ad of $\mathbf{S}_{\text{T}}$.

```
$(c_0, \mathsf{tiling})$ :
$y$ := fint(self);
$y$ := icheck(self);
$y$ := tcheck(self);
$y$ := fin(self);
$y$ := test($y$);
return(self).
```

```
$(c_0, \mathsf{icheck})$ :
$y$ := checkA1(self);
$y$ := checkA2(self);
return(self).
```

```
$(c_0, \mathsf{tcheck})$ :
$y$ := checkB1(self);
$y$ := checkB2(self);
$y$ := checkB3(self);
$y$ := checkB4_exchange(self);
$y$ := checkB5(self);
return(self).
```

Figure 4.5. Definition of method tiling.

of the first argument d of $f_{\mathbf{I}}$ (which denotes the row number) and $i_m \cdots i_{2m-1}$ be that of the second argument e (which denotes the column number). Assume that $o_0.a_{i_0}.\cdots.a_{i_{m-1}}.a_{i_m}.\cdots.a_{i_{2m-1}} \in \nu(c_j^t)$. Then, $f_{\mathbf{I}}(d,e) = t_j$.

Hereafter, we write $o.(a)^i$ instead of $o.\underbrace{a.\cdots.a}_{i}$.

Consider the execution of method tiling on $o_0 \in \nu(c_0)$ under $\mathbf{I}$. The tail object o_f is defined as $o_0.(a_0)^{2m}.(a_f)^{2m+1}.a_0^t.a.a_f$. As stated above, the a-value of o_f plays an important role. First, method fint sets $o_f.a$ to $o_f.a_t \in \nu(c'_{\text{fin,t}})$. Method icheck checks whether $\mathbf{I}$ has the ideal form. If not, $o_f.a$ is set to $o_f.a_f \in \nu(c'_{\text{fin,f}})$. Method tcheck checks whether $f_{\mathbf{I}}$ is an $n \times n$ tiling. If not, $o_f.a$ is set to $o_f.a_f$.

Method fin returns $o_f.a$. Since method test is defined only for class $c'_{\text{fin,f}}$, a type error occurs under $\mathbf{I}$ if and only if $\mathbf{I}$ has the ideal form and $f_{\mathbf{I}}$ is an $n \times n$ tiling.

In the following, we explain how icheck and tcheck work.

4.2.2 Method icheck

Let $o_a = o_0.(a_0)^{2m}.(a_f)^{2m}$ and $o_t = o_a.a_f.a_0^t.a$ (see Figure 4.2). Let $O_0 = \{o_0\}$ and for each $k \in [1, 2m]$,

$$O_k \;=\; \{o.\alpha \mid o \in O_{k-1} \text{ and } \alpha \in \{a_0, a_1\}\}.$$

Definition 4.3: $\mathbf{I}$ is *ideal* if

(A1) for every $o \in O_{2m-i}$ $(i \in [1, 2m])$,

$o.(a_0)^i.(a_f)^i = o.a_1.(a_0)^{i-1}.(a_f)^i$ and

$o.(a_0)^i.(a_f)^{i-1} \neq o.a_1.(a_0)^{i-1}.(a_f)^{i-1}$; and

(A2) for every $i \in [1, k-1]$, $o_a.a_f.a_i^t.a = o_t$. □

If $\mathbf{I}$ is ideal, then method tcheck can set $o_f.a$ to $o_f.a_f$ when it turns out that $f_{\mathbf{I}}$ is not an $n \times n$ tiling.

In what follows, we show that $\mathbf{I}$ is ideal if and only if $o_f.a \in \nu(c'_{\text{fin,t}})$ after executing icheck on o_0. Method $\mathsf{checkA1}$ checks whether $\mathbf{I}$ satisfies (A1). The algorithm of this method is shown in Figure 4.6.

Lemma 4.4: Let $o \in O_{2m-i}$ $(i \in [1, 2m])$ (see Figure 4.7). $o.(a_0)^i.(a_f)^i.a = o.(a_0)^i.(a_f)^i.a_t$ after the execution of $\mathsf{checkA1}_{2m-i}(o)$ if and only if for every $o_j \in O_{2m-j}$ $(j \in [1, i])$ reachable from o,

1. $o_j.(a_0)^j.(a_f)^j = o_j.a_1.(a_0)^{j-1}.(a_f)^j$; and
2. $o_j.(a_0)^j.(a_f)^{j-1} \neq o_j.a_1.(a_0)^{j-1}.(a_f)^{j-1}$.

Proof: The lemma is shown by induction on i. We show only the inductive step. Let $o \in O_{2m-i}$, $o' = o.a_0$, $o'' = o.a_1$, $o'_{i-1} = o'.(a_0)^{i-1}.(a_f)^{i-1}$, $o''_{i-1} = o''.(a_0)^{i-1}.(a_f)^{i-1}$, and $o'_i = o'_{i-1}.a_f$ (see Figure 4.7).

```
    $\mathsf{checkA1}_{2m-i}(o)$ $(i \in [1, 2m], o \in \nu(c_{2m-i}))$
    // define notations (see also Figure 4.7)
    let $o' = o.a_0$;
    let $o'' = o.a_1$;
    let $o'_{i-1} = o'.(a_0)^{i-1}.(a_f)^{i-1}$;
    let $o''_{i-1} = o''.(a_0)^{i-1}.(a_f)^{i-1}$;
    let $o'_i = o'_{i-1}.a_f$;
    // check recursively the smaller parts
1:  if $i \geq 2$ then
2:      $\mathsf{checkA1}_{2m-(i-1)}(o')$;
3:      $\mathsf{checkA1}_{2m-(i-1)}(o'')$;
    // check whether $o''_{i-1}.a_f = o'_i$
4:  $o'_i.a := o'_i.a_f$;
5:  $o''_{i-1}.a_f.a := o''_{i-1}.a_f.a_t$;
    // overwrite the checking results of the smaller parts
6:  if $i \geq 2$ then
7:      if $o'_{i-1}.a \in \nu(c'_{2m-i,f})$ then
8:          $o'_i.a := o'_i.a_f$;
9:      if $o''_{i-1}.a \in \nu(c'_{2m-i,f})$ then
10:         $o''_{i-1}.a_f.a := o''_{i-1}.a_f.a_f$;
    // check whether $o'_{i-1} \neq o''_{i-1}$
11: $o'_{i-1}.a := o'_{i-1}.a_t$;
12: $o''_{i-1}.a := o''_{i-1}.a_f$;
13: if $o'_{i-1}.a \in \nu(c'_{2m-i,f})$ then
14:     $o'_i.a := o'_i.a_f$;
    // store the final result to $o_f.a$
15: if $i = 2m$ and $o_a.a \in \nu(c_{\mathrm{com,f}})$ then
16:     $o_f.a := o_f.a_f$;
```

Figure 4.6. Algorithm of method checkA1.

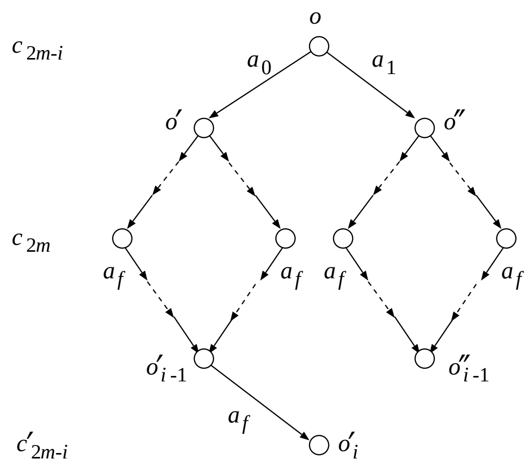


Figure 4.7. Method checkA1.

[*if part*] Suppose that both 1 and 2 hold for every $o_j \in O_{2m-j}$ ($j \in [1, i]$) reachable from o. This implies that $o''_{i-1}.a_f = o'_i$ and $o'_{i-1} \neq o''_{i-1}$. Consider the execution of checkA1$_{2m-i}(o)$. Since $o''_{i-1}.a_f = o'_i$, $o'_i.a$ is set to $o'_i.a_t \in \nu(c'_{2m-i-1,t})$ in line 5. From the inductive hypothesis, $o'_{i-1}.a = o'_{i-1}.a_t$ and $o''_{i-1}.a = o''_{i-1}.a_t$ after the execution of line 3, so the conditions of lines 7 and 9 do not hold. Since $o'_{i-1} \neq o''_{i-1}$, $o'_{i-1}.a = o'_{i-1}.a_t \in \nu(c'_{2m-i,t})$ after the execution of line 12, the condition of line 13 does not hold. Thus, $o'_i.a = o'_i.a_t$ after the execution of checkA1$_{2m-i}(o)$.

[*only if part*] Suppose that $o'_i.a = o'_i.a_t$ after the execution of checkA1$_{2m-i}(o)$. Then, by the algorithm and the inductive hypothesis,

- $o''_{i-1}.a_f = o'_i$;
- $o'_{i-1} \neq o''_{i-1}$; and
- 1 and 2 hold for every $o_j \in O_{2m-j}$ ($j \in [1, i-1]$) reachable from o' or o''.

Thus, 1 and 2 hold for every $o_j \in O_{2m-j}$ ($j \in [1, i]$) reachable from o. □
By letting $i = 2m$, we can conclude that $o_a.a \in \nu(c_{\text{com,f}})$ (and therefore, $o_f.a = o_f.a_f$ by line 16) after the execution of checkA1(o_0) if and only if $\mathbf{I}$ does not satisfy (A1).

Method checkA2 checks whether $\mathbf{I}$ satisfies (A2). Whenever $\mathbf{I}$ satisfies (A1), $o_f.a = o_f.a_f$ after the execution of checkA2(o_0) if and only if $\mathbf{I}$ does not satisfy (A2). This method can be constructed easily.

4.2.3 Method tcheck

In this section, we show that the following lemma holds.

Lemma 4.5: Suppose that $\mathbf{I}$ is ideal. Then, after executing tcheck on o_0, $o_f.a \in \nu(c'_{\text{fin,t}})$ if and only if

(B1) $f_{\mathbf{I}}$ satisfies (t-a) in Definition 4.2;

(B2) for every $o \in O_{2m}$, $o \in \nu(c_0^t) \cup \cdots \cup \nu(c_{k-1}^t)$;

(B3) $f_{\mathbf{I}}$ satisfies (t-b) in Definition 4.2;

(B4) $\mathbf{I}$ satisfies the *exchangeability condition* explained below. Informally, the exchangeability condition means that $\mathbf{I}$ can be easily transformed into $\mathbf{I}'$ such that $o_0.a_{i_0}.\cdots.a_{i_{m-1}}.a_{i_m}.\cdots.a_{i_{2m-1}}$ in $\mathbf{I}$ is equal to $o_0.a_{i_m}.\cdots.a_{i_{2m-1}}.a_{i_0}.\cdots.a_{i_{m-1}}$ in $\mathbf{I}'$. Figures 4.8 and 4.9 are examples of $\mathbf{I}$ and $\mathbf{I}'$;

(B5) $f_{\mathbf{I}}$ satisfies (t-c) in Definition 4.2. □

(B1)–(B3) and (B5) together mean that $f_{\mathbf{I}}$ is a tiling, while (B4) is a technical condition to be able to check (B5) in polynomial time. To check (B1)–(B3) is comparatively easy if $\mathbf{I}$ has the ideal form. However, to check (B5), it may take exponential time without any transformation of $\mathbf{I}$ such as (B4).

Method checkB1 checks whether $\mathbf{I}$ satisfies (B1). $o_f.a$ is set to $o_f.a_f$ if and only if $o_0.(a_0)^{2m} \notin \nu(c_0^t)$.

Method checkB2 checks whether $\mathbf{I}$ satisfies (B2). For every $o \in O_{2m}$, if $o \notin \nu(c_0^t) \cup \cdots \cup \nu(c_{k-1}^t)$, that is, o does not represent any tile in T, then $o_f.a$ is set to $o_f.a_f$.

Method checkB3 checks whether $\mathbf{I}$ satisfies (B3). When checkB3 is invoked on $o \in O_{2m-i}$ for some $i \in [1, m]$, it checks whether the two tiles represented by $o.a_0.(a_1)^{i-1}$ and $o.a_1.(a_0)^{i-1}$ have compatibility given by H. By invoking checkB3

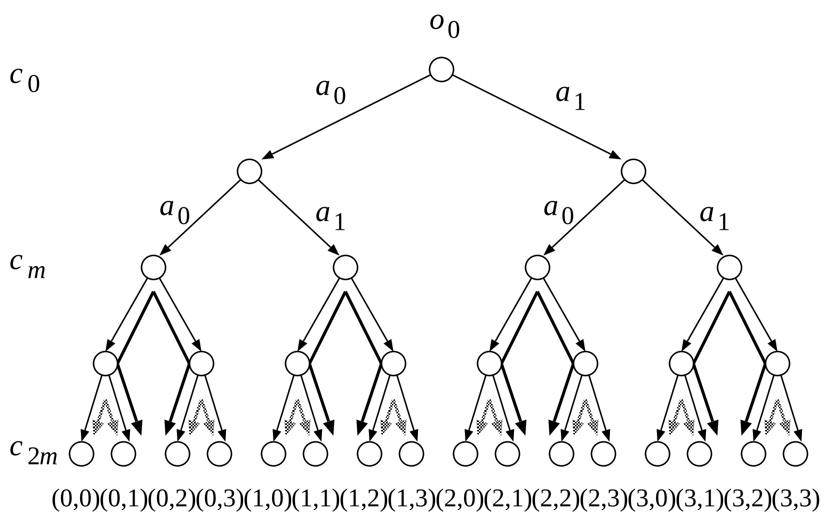


Figure 4.8. Method checkB3.

on every $o \in O_{2m-i}$ for each $i \in [1, m]$, it can be checked whether $\mathbf{I}$ satisfies (B3) (see Figure 4.8).

Here we explain how to check whether two tiles represented by o', $o'' \in O_{2m}$ have compatibility given by H. First, if $o' \in \nu(c_x^t)$, then $o_a.a_f.a$ is set to $o_a.a_f.a_x^t \in \nu(c_x^{t'})$. Next, if $o'' \in \nu(c_y^t)$, then method harg_2^y is invoked on $o_a.a_f.a$. $Impl(c_x^{t'}, \mathsf{harg}_2^y)$ is defined so that $o_f.a$ is set to $o_f.a_f$ if and only if $(t_x, t_y) \notin H$. Thus, $o_f.a$ is set to $o_f.a_f$ during the execution of checkB3(o_0) if and only if $\mathbf{I}$ does not satisfy (B3).

Method checkB4_exchange updates $\mathbf{I}$ so that method checkB5, which is defined similarly to checkB3, can check whether $\mathbf{I}$ satisfies (B5) (see Figure 4.9). More precisely, it transforms $\mathbf{I}$ into $\mathbf{I}'$ so that for every $o_{2m} \in O_{2m}$, if $o_{2m} = o_0.a_{i_0}.\cdots.a_{i_{m-1}}.a_{i_m}.\cdots.a_{i_{2m-1}}$ under $\mathbf{I}$, then $o_{2m} = o_0.a_{i_m}.\cdots.a_{i_{2m-1}}.a_{i_0}.\cdots.a_{i_{m-1}}$ under $\mathbf{I}'$. That is, $i_0 \cdots i_{m-1}$ (the row number) and $i_m \cdots i_{2m-1}$ (the column number) are exchanged. Without checkB4_exchange, it seems impossible to construct a method in polynomial time which checks the compatibility defined by V.

The transformation algorithm is shown in Figure 4.10. Let us consider how the path expression from o_0 to o_{2m} is transformed. Let $o_{m-1} = o_0.a_{i_0}.\cdots.a_{i_{m-2}}$. First, $o_{m-1}.a_{i_{m-1}}.a_{i_m}$ and $o_{m-1}.a_{i_m}.a_{i_{m-1}}$ are swapped in line 4 when $k = 1$ and

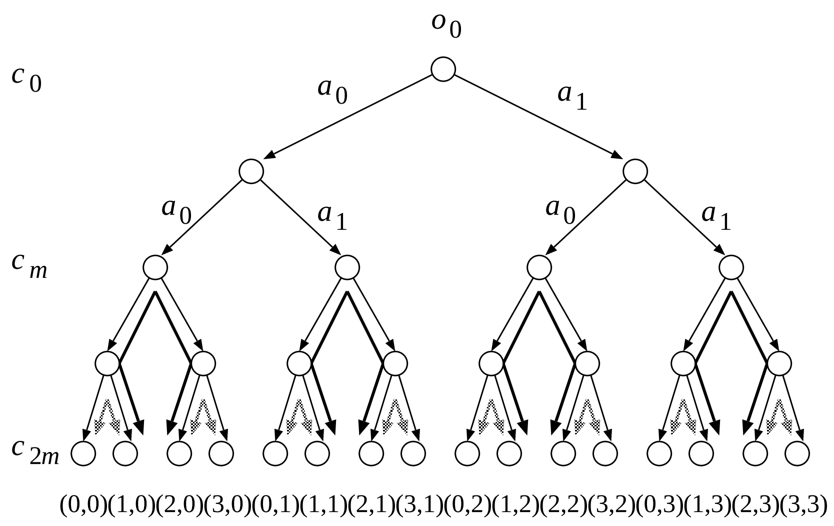


Figure 4.9. Method checkB5.

$l = 0$. Thus,

$$o_{2m} = o_{m-1}.a_{i_m}.a_{i_{m-1}}.a_{i_{m+1}}.\cdots.a_{i_{2m-1}}.$$

Note that the position of $a_{i_{m-1}}$ in the path expression is shifted to right by one. Then, the same swap is done for $o_m = o_{m-1}.a_{i_m}$ when $l = 1$, and the position of $a_{i_{m-1}}$ is shifted to right by one again. By repeating this until $l = m - 1$,

$$o_{2m} = o_{m-1}.a_{i_m}.\cdots.a_{i_{2m-1}}.a_{i_{m-1}}.$$

Next, k is incremented. Let $o_{m-2} = o_0.a_{i_0}.\cdots.a_{i_{m-3}}$. $o_{m-2}.a_{i_{m-2}}.a_{i_m}$ and $o_{m-2}.a_{i_m}.a_{i_{m-2}}$ are swapped similarly when $l = 0$, and thus,

$$o_{2m} = o_{m-2}.a_{i_m}.a_{i_{m-2}}.a_{i_{m+1}}.\cdots.a_{i_{2m-1}}.a_{i_{m-1}}.$$

In a similar way to the above case, by repeating this until $l = m-1$, the position of $a_{i_{m-2}}$ is shifted to right by m, and thus,

$$o_{2m} = o_{m-2}.a_{i_m}.\cdots.a_{i_{2m-1}}.a_{i_{m-2}}.a_{i_{m-1}}.$$

Similarly, by doing these swaps for all k and l,

$$o_{2m} = o_0.a_{i_m}.\cdots.a_{i_{2m-1}}.a_{i_0}.\cdots.a_{i_{m-1}}.$$

```
    let $o_f = o.(a_0)^{m+k-l}.(a_f)^{2m}.a_0^t.a.a_f$;
1:  for $k := 1$ to $m$ do
2:      for $l := 0$ to $m - 1$ do
3:          for each $o \in O_{m-k+l}$
4:              swap $o.a_0.a_1$ and $o.a_1.a_0$
```

Figure 4.10. Transformation algorithm.

Unfortunately, the swap operation of line 4 is impossible in general because of the ability of update schemas. However, if **I** satisfies the exchangeability condition, the swap operation becomes possible. Actually, method checkB4_exchange checks (B4) (i.e., exchangeability condition) and transforms **I** simultaneously. If it turns out that **I** does not satisfy (B4), then method checkB4_exchange sets $o_f.a$ to $o_f.a_f$. The exchangeability condition is complicated, so we omit its formal definition here. See Reference [20] for details.

Method checkB5 sets $o_f.a$ to $o_f.a_f$ if and only if **I** does not satisfy (B5).

Thus, Lemma 4.5 holds.

From Theorems 4.1 and 4.2, the type-consistency problem for recursion-free acyclic schemas is shown to be coNEXPTIME-complete.

4.3 The Upper Bound for Retrieval Acyclic Schemas

In this and the following sections, we show that the type-consistency problem for retrieval acyclic schemas is PSPACE-complete. First, we show that the type-consistency problem for retrieval acyclic schemas is in PSPACE.

Let **S** be a retrieval acyclic schema and **I** be an arbitrary instance of **S**. Let o_0 be an object in **I**. For **I** and o_0, an instance which satisfies the following properties is called a *tree-shaped* instance $\mathbf{I}_t$ with root o_0:

- Let p be a path starting from o_0 in **I** and o be the end point of p. Let o' be the end point of p in $\mathbf{I}_t$. Then, $cl(o) = cl(o')$; and

- For any o and o' which are the end points of distinct paths p and p' from o_0 in $\mathbf{I}_t$, $o \neq o'$.

Since $\mathbf{S}$ is a retrieval schema, it can be shown that if the execution of $m_0(o_0)$ under $\mathbf{I}$ causes a type error, then that of $m_0(o_0)$ under $\mathbf{I}_t$ also causes a type error. Thus, only tree-shaped instances need to be considered to decide the consistency of $\mathbf{S}$. So we assume $\mathbf{I}$ is a tree in the following discussion.

A nondeterministic polynomial-space algorithm for deciding type-consistency of a retrieval acyclic schema $\mathbf{S}$ is shown in Figure 4.11. The basic idea is that guess an object o_0, a method m_0, and an inconsistent instance $\mathbf{I}$, and then simulate $m_0(o_0)$ on $\mathbf{I}$. However, to check the consistency of $\mathbf{S}$ in PSPACE, we cannot guess an entire instance of $\mathbf{S}$ because it needs an exponential space to do so. Instead, the proposed algorithm decides the consistency by guessing one path of $\mathbf{I}$ in an on-line manner and executing $m_0(o_0)$ on the path until all the paths of $\mathbf{I}$ are guessed.

For a path p in $\mathbf{I}$, we define a *pseudo instance* $\mathbf{I}_p$ as follows: (i) the objects on p are the same as $\mathbf{I}$ and (ii) the other parts are all replaced with ω, where ω is a special symbol. A method execution on a pseudo instance is done in the same way as that on an ordinary instance except for method invocations on ω. If the control reaches a sentence of the form "$y := m(y')$" and ω is assigned to y', then ω is assigned to y without executing m.

Since $\mathbf{S}$ is a retrieval schema, it can be shown that an execution is nonterminating if and only if m is invoked on o during the execution of $m(o)$ for some m and o. Nonterminating executions can be detected in this way, and then the procedure stops.

Now, assume that a type error occurs on some $\mathbf{I}_p$ and a nonterminating execution is detected on another $\mathbf{I}_{p'}$. Let st_p and $st_{p'}$ be the runtime stacks when the executions are aborted and found nonterminating, respectively. If the runtime stack of the execution of $m_0(o_0)$ on $\mathbf{I}$ gets equal to $st_{p'}$ before st_p, then the execution falls into a loop and no type error occurs. Therefore, we have to know only the runtime stack that is reachable first among all the st_p's. Procedure Pre provides the order among runtime stacks. Procedure Pre takes two arguments, (i) the runtime stack of the previous execution and (ii) the runtime stack that is reachable first until then, and returns the one that is reachable first.

```
Procedure DEC-CONSISTENCY(S)
begin
guess an object $o_0$ and $m_0 \in Meth$;
$st$ := undefined;
repeat
    guess a path $p$ from $o_0$ of I in an on-line manner;
    execute $m_0(o_0)$ on $\mathbf{I}_p$;
    if the execution is aborted due to a type error
        or it is found nonterminating then
        $st$ := PRE($st$, the runtime stack of the execution);
until all the paths of I are guessed;
if $st$ causes a type error then
    output "S is inconsistent.";
else
    output "The guess failed.";
end
```

Figure 4.11. Procedure DEC-CONSISTENCY.

We show the correctness of this procedure. Consider the executions of $m(o)$ on $\mathbf{I}$ and $\mathbf{I}_p$, where o is an object in p. Then, the following statements hold.

- If the value of y' is an object in p when executing some sentence of $Impl^*(cl(o), m)$ on $\mathbf{I}$, then the value of y' is the same object when executing the same sentence on $\mathbf{I}_p$; and
- If the value of y' is an object which is not in p when executing some sentence of $Impl^*(cl(o), m)$ on $\mathbf{I}$, then the value of y' is ω when executing the same sentence on $\mathbf{I}_p$.

Suppose that a type error occurs when trying to invoke some method on an object in p during the execution of $m_0(o_0)$ on $\mathbf{I}$. From the above statements, when executing $m_0(o_0)$ on $\mathbf{I}_p$, a type error occurs at the same sentence because the sequences of method invocations on objects in p are the same. For any $p' \neq p$

of $\mathbf{I}$, the execution of $m_0(o_0)$ on $\mathbf{I}_{p'}$ does not cause a type error at least until the control reaches the sentence. Thus, the runtime stack of the execution on $\mathbf{I}_p$ is stored in st after finishing the execution for all the $\mathbf{I}_p$'s. The opposite direction also holds, that is, if st after the execution for all the $\mathbf{I}_p$'s causes a type error, then a type error occurs on $\mathbf{I}$.

Theorem 4.3: The type-consistency problem for retrieval acyclic schemas is in PSPACE. □

4.4 The Lower Bound for Retrieval Acyclic Schemas

Theorem 4.4: The type-consistency problem for retrieval acyclic schemas is PSPACE-hard even if the height of the class hierarchy is at most one.

Proof: We show that the problem is PSPACE-hard by a reduction from the well-known QBF problem.

Given a quantified boolean formula F, we construct a retrieval acyclic schema $\mathbf{S}_\mathrm{F}$ such that $\mathbf{S}_\mathrm{F}$ is inconsistent if and only if F is true. Let

$$F = \exists x_1 \forall x_2 \cdots \exists x_{2k+1} \varphi,$$

where k is an integer, $x_1, \ldots, x_{2k+1}$ are variables, and φ is a boolean formula in 3CNF on $x_1, \ldots, x_{2k+1}$. Let n be the number of clauses in φ. Let C_i ($i \in [1, n]$) denote the i-th clause in φ. Let $l_{i,j}$ ($j \in [1, 3]$) denote the j-th literal in C_i. Let $ind_{i,j}$ denote the index of the variable appearing in $l_{i,j}$. Without loss of generality, we assume that $ind_{i,j} \leq ind_{i,j'}$ if $j \leq j'$.

An example of a database instance of $\mathbf{S}_\mathrm{F}$ is shown in Figure 4.12. Assignments to existentially quantified variables are determined by a database instance. Each path from o_1 to a leaf represents one possible assignment to all variables. We define $\mathbf{S}_\mathrm{F}$ so that:

- a type error occurs when φ is satisfied for every path (that is, F is true), and

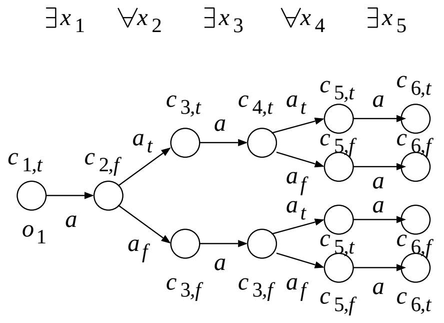


Figure 4.12. Example of a database instance of $\mathbf{S}_{\mathrm{F}}$.

- the execution is nonterminating (therefore, no type error occurs) when φ is not satisfied for some path (that is, F is false).

Formally, $\mathbf{S}_{\mathrm{F}} = (C, \preceq, \mathit{Attr}, \mathit{Ad}, \mathit{Meth}, \mathit{Impl})$ is constructed as follows:

- $C = \{c_{1,t}, c_{2,t}, \ldots, c_{2k+2,t}, c_{2,f}, c_{3,f}, \ldots, c_{2k+2,f}\}$;
- $c_{2i,f} \preceq c_{2i,t}$ for each $i \in [1, k+1]$;
- $\mathit{Ad}(c_{2i-1,t}, a) = c_{2i,t}$ for each $i \in [1, k+1]$,
  $\mathit{Ad}(c_{2i,t}, a_t) = c_{2i+1,t}$ for each $i \in [1, k]$,
  $\mathit{Ad}(c_{2i,t}, a_f) = c_{2i+1,f}$ for each $i \in [1, k]$; and
- Definition of *Impl* is shown in Figure 4.13. The underlined part is a macro notation, and all of them can be expanded when F is reduced to $\mathbf{S}_{\mathrm{F}}$. Note that method test is undefined for all classes.

Let $\mathbf{I}$ be a database instance of $\mathbf{S}_{\mathrm{F}}$. Consider the execution of method qbf on $o_1 \in \nu(c_{1,t})$ under $\mathbf{I}$. The a-value of an object in $\nu(c_{2i-1,t})$ or $\nu(c_{2i-1,f})$ represents an assignment to an existentially quantified variable x_{2i-1}. If it is an object in $\nu(c_{2i,t})$ (resp. $\nu(c_{2i,f})$), then true (resp. false) is assigned to x_{2i-1}. Attribute a_t (resp. a_f) of an object in $\nu(c_{2i,t})$ or $\nu(c_{2i,f})$ represents that true (resp. false) is assigned to an universally quantified variable x_{2i}. Thus, each path from o_1 to an

```
($c_{1,t}$, qbf) :
for each $i \in [1, n]$
    $y$ := clause$_{i,1}$(self);
$y$ := test(self);
return(self).
```

for each i, j, l s.t. $2l \leq ind_{i,j}$

```
($c_{2l,t}$, clause$_{i,j}$) :
$y$ := clause$_{i,j}$(self.$a_t$);
$y$ := clause$_{i,j}$(self.$a_f$);
return(self).
```

for each i, j, l s.t. $2l + 1 \leq ind_{i,j}$

```
($c_{2l+1,t}$, clause$_{i,j}$) :
$y$ := clause$_{i,j}$(self.$a$);
return(self).
```

```
($c_{2l+1,f}$, clause$_{i,j}$) :
$y$ := clause$_{i,j}$(self.$a$);
return(self).
```

for each i, j, l s.t. $l = ind_{i,j}$ and $l_{i,j}$ is positive

```
($c_{l+1,t}$, clause$_{i,j}$) :
return(self).
```

```
($c_{l+1,f}$, clause$_{i,j}$) :
$y$ := clause$_{i,j+1}$(self);
return(self).
```

for each i, j, l s.t. $l = ind_{i,j}$ and $l_{i,j}$ is negative

```
($c_{l+1,t}$, clause$_{i,j}$) :
$y$ := clause$_{i,j+1}$(self);
return(self).
```

```
($c_{l+1,f}$, clause$_{i,j}$) :
return(self).
```

for each i, l s.t. $l = ind_{i,3} + 1$

```
($c_{l,t}$, clause$_{i,4}$) :
loop forever
```

```
($c_{l,f}$, clause$_{i,4}$) :
loop forever
```

Figure 4.13. Definition of *Impl* of $\mathbf{S}_{\mathrm{F}}$.

object in $\nu(c_{2k+2,t})$ or $\nu(c_{2k+2,f})$ defines one assignment. For example, a database instance shown in Figure 4.12 represents four assignments (from top to bottom):

$$\begin{aligned}
\sigma_1(x_1, x_2, x_3, x_4, x_5) &= (0, 1, 1, 1, 1),\\
\sigma_2(x_1, x_2, x_3, x_4, x_5) &= (0, 1, 1, 0, 0),\\
\sigma_3(x_1, x_2, x_3, x_4, x_5) &= (0, 0, 0, 1, 0),\\
\sigma_4(x_1, x_2, x_3, x_4, x_5) &= (0, 0, 0, 0, 1).
\end{aligned}$$

Let $\mathcal{A}_{\mathbf{I}}$ be the set of such assignments under $\mathbf{I}$.

Method qbf checks whether all the clauses are satisfied by every assignment $\mathcal{A}_{\mathbf{I}}$. Method $\mathsf{clause}_{i,j}$ checks whether $l_{i,j}$ is true under each assignment. If so, it just executes return to evaluate $l_{i+1,1}$. Otherwise, it invokes $\mathsf{clause}_{i,j+1}$ to check whether $l_{i,j+1}$ is true. Thus, the invocation of $\mathsf{clause}_{i,4}$ means that C_i is false. In this case, the execution is nonterminating (therefore, no type error occurs). When all the clauses are true by every assignment in $\mathcal{A}_{\mathbf{I}}$, method test is invoked, and then a type error occurs. Note that there exists such $\mathbf{I}$ that all the clauses are satisfied by every assignment in $\mathcal{A}_{\mathbf{I}}$ if and only if F is true. $\square$

From Theorems 4.3 and 4.4, the type-consistency problem for retrieval acyclic schemas is shown to be PSPACE-complete.

Chapter 5

Conclusions

In this thesis, we have discussed the complexity of the type-consistency problem for several subclasses of update schemas.

In Chapter 3, we have shown that the type-consistency problems for retrieval and terminating schemas are both undecidable. By these results, the complexity of the type-consistency problem for all the subclasses introduced by Reference [12] have been obtained. Moreover, by comparing the results, it turns out that non-flatness of the class hierarchy, recursion, and update operations each make the problem difficult. When we classify update schemas in view of these factors, the problem is undecidable or intractable for most of practical cases.

In Chapter 4, we have introduced another subclass of update schemas, called acyclic schemas, which is practical and for which consistency is decidable. We have shown that the type-consistency problem for acyclic schemas is in coNEXP-TIME in monadic cases. The problem remains to be decidable in polyadic cases because a type error can be detected by guessing an instance and simulating the execution on the instance, as in Theorem 4.1. It is open whether the problem for polyadic acyclic schemas is still in coNEXPTIME. Furthermore, we have introduced two subclasses of acyclic schemas, called recursion-free acyclic schemas and retrieval acyclic schemas, and shown that the type-consistency problems for these two subclasses are coNEXPTIME-hard and PSPACE-complete, respectively.

To propose other subclasses of update schemas for which the type-consistency problem is solvable more efficiently is the future work. 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. Computer and System Sciences, vol.51, no.3, pp.433–455, Dec. 1995.

[3] S. Abiteboul, P. Kanellaskis, and E. Waller, "Method schemas," 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," Proc. 6th Conf. on Object-Oriented Programming Systems, Languages, and Applications, pp.113–128, Oct. 1991.

[5] R. Ahad, J. Davis, S. Gower, P. Lyngbaek, A. Marynowski, and E. Onuegbe, "Supporting access control in an object-oriented database language," Proc. 3rd Int'l Conf. on Extending Database Technology, LNCS 580, pp.184–200, Mar. 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, vol.5, no.2, pp.133–150, Apr. 1996.

[7] E. Bertino, "Data hiding and security in object-oriented databases," Proc. 8th IEEE Int'l Conf. on Data Engineering, pp.338–247, Feb. 1992.

[8] C. Chambers and G. T. Leavens, "Typechecking and modules for multimethods," ACM Trans. on Programming Languages and Systems, vol.17, no.6, pp.805–843, Nov. 1995.

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

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

[11] 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.

[12] Y. Ishihara, H. Seki, and M. Ito, "Type-consistency problems for queries in object-oriented databases," Proc. 6th Int'l Conf. on Database Theory, LNCS 1186, pp.364–378, Jan. 1997.

[13] J. Leeuwen ed., "Algorithms and Complexity," North-Holland, 1990.

[14] N. Oxhϕj, J. Palsberg, and M. I. Schwartzbach, "Making type inference practical," Proc. European Conf. on Object-Oriented Programming, LNCS 615, pp.329–349, June 1992.

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

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

[17] H. Seki, Y. Ishihara, and H. Dodo, "Testing type consistency of method schemas," IEICE Transactions on Information and Systems, vol.E81-D, no.3, pp.278–287, March 1998.

[18] H. Seki, Y. Ishihara, and M. Ito, "Authorization analysis of queries in object-oriented databases," Proc. 4th Int'l Conf. on Deductive and Object-Oriented Databases, LNCS 1013, pp.521–538, Dec. 1995.

[19] E. Waller, "Schema updates and consistency," Proc. 2nd Int'l Conf. on Deductive and Object-Oriented Databases, LNCS 566, pp.167–188, Dec. 1991.

[20] J. Yokouchi, "Complexity of the type-consistency problem for acyclic object-oriented database schemas," Master's Thesis, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-MT9851125, Feb. 2000 (in Japanese).

```
$(c_\mathrm{f}, \mathsf{test})$ :
$\mathsf{return}(\mathsf{self})$.
```

Figure A.1. Definition of method test.

```
$(c_\mathrm{t}, \mathsf{get\_ws})$ :
$\mathsf{self}.a_\mathrm{ws} := \mathsf{true}$;
$y := \mathsf{get\_ws}'(\mathsf{self})$;
$\mathsf{return}(\mathsf{self})$.
```

```
$(c_\mathrm{t}, \mathsf{get\_ws}')$ :
$\underline{y := \mathsf{self}\ \mathsf{if}\ \mathsf{self}.a_\mathrm{ws} = \mathsf{true}}$
    $\underline{\text{and } y := \mathsf{self}.a'_\mathrm{t} \text{ otherwise}}$;
$y' := \mathsf{if\_then}[a_\mathrm{ws}, \mathsf{get\_ws}''](\mathsf{self})$;
$\mathsf{return}(y)$.
```

```
$(c, \mathsf{get\_ws})$ :
$\mathsf{return}(\mathsf{self})$.
```

```
$(c, \mathsf{get\_ws}')$ :
$\mathsf{return}(\mathsf{self}.a'_\mathrm{t})$.
```

```
$(c, \mathsf{get\_ws}'')$ :
$\mathsf{return}(\mathsf{self})$.
```

```
$(c_\mathrm{t}, \mathsf{get\_ws}'')$ :
$\mathsf{self}.a_\mathrm{ws} := \mathsf{false}$;
$y := \mathsf{get\_ws}'(\mathsf{self}.a_\Rightarrow)$;
$\mathsf{self}.a_\Rightarrow := y$;
$\mathsf{self}.a_\mathrm{ws} := \mathsf{true}$;
$\mathsf{self}.a'_\mathrm{ws} := \mathsf{true}$;
$\underline{\text{set } \mathsf{self}.\vec{a} \text{ to } \langle \bot, B \rangle}$;
$\mathsf{return}(\mathsf{self})$.
```

Figure A.2. Definition of method get_ws.

Appendix

A Complete Proof of Theorem 3.2

Let M be a Turing machine and $x = x_1 \cdots x_n$ an input string for M. We abbreviate $\mathsf{self}.a := \mathsf{self}$ and $\mathsf{self}.a := \mathsf{self}.a_\mathrm{f}$ to $\mathsf{self}.a := \mathsf{true}$ and $\mathsf{self}.a := \mathsf{false}$, respectively. Methods test, get_ws, initws, step, accept are defined as shown in Figures A.1–A.5, respectively.

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

Lemma A.1: 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 $\mathsf{get\_ws}$ for o_1 is terminating under $\mathbf{I}$.

Proof: If $o_1 \in \nu(c'_{\mathrm{t}}) \cup \nu(c_{\mathrm{f}}) \cup \nu(c)$, then the execution is terminating since $(c, \mathsf{get\_ws})$ is executed for o_1. Thus in the following we consider the remaining case, i.e., $o_1 \in \nu(c_{\mathrm{t}})$. First of all, by the first line of $(c_{\mathrm{t}}, \mathsf{get\_ws})$, $o_1.a_{\mathrm{ws}}$ is set to true. Then, $\mathsf{get\_ws}'$ is invoked on o_1. By the second line of $(c_{\mathrm{t}}, \mathsf{get\_ws}')$, $\mathsf{get\_ws}''$ is invoked on o_1 since $o_1.a_{\mathrm{ws}}$ is true. By $(c_{\mathrm{t}}, \mathsf{get\_ws}'')$, $\mathsf{get\_ws}''$ sets $o_1.a_{\mathrm{ws}}$ false and recursively invokes $\mathsf{get\_ws}'$ on $o_1.a_{\Rightarrow}$.

Consider the case that $\mathsf{get\_ws}'$ is recursively invoked on an object o. There are three cases to be considered:

(1) If $o \in \nu(c'_{\mathrm{t}}) \cup \nu(c_{\mathrm{f}}) \cup \nu(c)$, then the recursive invocation of $\mathsf{get\_ws}'$ terminates since $(c, \mathsf{get\_ws}')$ is executed for o.

(2) If $o \in \nu(c_{\mathrm{t}})$ and $o.a_{\mathrm{ws}}$ is false, then no more recursive invocation occurs from the definition of $(c_{\mathrm{t}}, \mathsf{get\_ws}')$.

(3) If $o \in \nu(c_{\mathrm{t}})$ and $o.a_{\mathrm{ws}}$ is true, then $\mathsf{get\_ws}''$ is invoked on o by the second line of $(c_{\mathrm{t}}, \mathsf{get\_ws}')$. Method $\mathsf{get\_ws}''$ sets $o.a_{\mathrm{ws}}$ false and recursively invokes $\mathsf{get\_ws}'$ on $o.a_{\Rightarrow}$. Thus, every time $\mathsf{get\_ws}'$ is recursively invoked, the number of objects o such that $o.a_{\mathrm{ws}}$ 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 $\mathsf{get\_ws}$ 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 A.2: $\mathbf{S}_{M,x}$ is terminating. □

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

Lemma A.3: Let $\mathbf{I} = (\nu, \mu)$ be an arbitrary database instance of $\mathbf{S}_{M,x}$, and $o_1 \in \nu(c_{\mathrm{t}})$ be an arbitrary object. After the execution of $\mathsf{get\_ws}$ for o_1 under $\mathbf{I}$, there exists a positive integer k which satisfies the following condition (C1):

(C1-1) $o_1 \in \nu(c_{\mathrm{t}})$, $o_i.a_{\Rightarrow} = o_{i+1} \in \nu(c_{\mathrm{t}})$ $(i \in [1, k-1])$, and $o_k.a_{\Rightarrow} = o_{k+1} \in \nu(c'_{\mathrm{t}})$;

(C1-2) $o_i.a_{\mathrm{ws}} = o_i.a'_{\mathrm{ws}} = \mathsf{true}$ $(i \in [1, k])$;

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

Proof: Suppose that $\mathsf{get\_ws'}$ is invoked k times by the second line of $(c_{\mathrm{t}}, \mathsf{get\_ws''})$ during the complete execution of $\mathsf{get\_ws}$ 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 $(c_{\mathrm{t}}, \mathsf{get\_ws})$, $\mathsf{get\_ws'}$ is invoked on o_1. Since $o_1.a_{\mathrm{ws}}$ is true by the first line of $(c_{\mathrm{t}}, \mathsf{get\_ws})$, $\mathsf{get\_ws''}$ is invoked on o_1 by the second line of $(c_{\mathrm{t}}, \mathsf{get\_ws'})$. Then, by the second line of $(c_{\mathrm{t}}, \mathsf{get\_ws''})$, $\mathsf{get\_ws'}$ is invoked on $o_1.a_{\Rightarrow} = o_2$. Thus $k \geq 1$.

Next, we prove (C1-1). Consider the i-th invocation $(i \in [1, k-1])$ of $\mathsf{get\_ws'}$ from the second line of $(c_{\mathrm{t}}, \mathsf{get\_ws''})$. Let o_{i+1} be the self object of the invocation. Note that $o_{i+1} \in \nu(c_{\mathrm{t}})$ and $o_{i+1}.a_{\mathrm{ws}}$ is true since $i < k$ (see the condition (3) in the proof of Lemma A.1). By the first and third lines of $(c_{\mathrm{t}}, \mathsf{get\_ws'})$, the returned value of this invocation is o_{i+1}. Therefore, by the second and third lines of $(c_{\mathrm{t}}, \mathsf{get\_ws''})$, it holds that $o_i.a_{\Rightarrow} = o_{i+1} \in \nu(c_{\mathrm{t}})$. Next, consider the k-th invocation of $\mathsf{get\_ws'}$, 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 A.1 holds. If (1) holds, then $o.a'_{\mathrm{t}}$ is returned as the returned value of this invocation since $(c, \mathsf{get\_ws'})$ is executed for o (see also Figure 3.11(1)). If (2) holds, then $o.a'_{\mathrm{t}}$ is returned by the first and third lines of $(c_{\mathrm{t}}, \mathsf{get\_ws'})$ (see also Figure 3.11(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 $(c_{\mathrm{t}}, \mathsf{get\_ws''})$. 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 $(c_{\mathrm{t}}, \mathsf{get\_ws''})$. □

The following lemma holds evidently from the definition of method initws (see Figure A.3).

Lemma A.4: Suppose that $\mathbf{I} = (\nu, \mu)$ satisfies condition (C1) for some k $(k \geq 1)$. Then, after the execution of initws for o_1 under $\mathbf{I}$, the following condition (C2) holds:

```
$(c_t, \mathsf{initws})$ :
$\underline{\text{set } \mathsf{self}.\vec{a} \text{ to } \langle q_0, \rhd \rangle}$;
$y := \mathsf{initws1}(\mathsf{self}.a_{\Rightarrow})$;
$\mathsf{return}(\mathsf{self})$.

$(c, \mathsf{initws})$ :
$\mathsf{return}(\mathsf{self})$.

$(c_t, \mathsf{initws1})$ :
$\underline{\text{set } \mathsf{self}.\vec{a} \text{ to } \langle \perp, x_1 \rangle}$;
$y := \mathsf{initws2}(\mathsf{self}.a_{\Rightarrow})$;
$\mathsf{return}(\mathsf{self})$.

$(c, \mathsf{initws1})$ :
$\mathsf{return}(\mathsf{self})$.

        ⋮                                ⋮

$(c_t, \mathsf{initws}n)$ :
$\underline{\text{set } \mathsf{self}.\vec{a} \text{ to } \langle \perp, x_n \rangle}$;
$\mathsf{return}(\mathsf{self})$.

$(c, \mathsf{initws}n)$ :
$\mathsf{return}(\mathsf{self})$.
```

Figure A.3. Definition of method initws.

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

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

(C2-3) For each $i \in [1, k]$, $o_i.\vec{a}$ 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 Figure A.4), 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 A.5: Suppose that $\mathbf{I} = (\nu, \mu)$ satisfies the following condition (C3) for some k ($k \geq 1$):

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

(C3-2) $o_i.a'_{\mathrm{ws}} = \mathsf{true}$ ($i \in [1, k]$);

```
$(c_{\rm t}, \mathsf{step})$ :
$y$ := if_then[$a_{\rm cont}$, step′](self);
return(self).
```

```
$(c_{\rm t}, \mathsf{step}')$ :
$y$ := if_then[$a_{\rm ws}$, step″](self);
return(self).
```

```
$(c_{\rm t}, \mathsf{step}'')$ :
self.$a_{\rm ws}$ := false;
$y$ := delta(self);
$y$ := step(self.$a_{\Rightarrow}$);
self.$a_{\rm ws}$ := true;
return(self).
```

```
$(c, \mathsf{step})$ :
return(self).
```

```
$(c, \mathsf{step}')$ :
return(self).
```

```
$(c, \mathsf{step}'')$ :
return(self).
```

```
$(c_{\rm t}, \mathsf{delta})$ :
$y$ := if_then[$a'_{\rm ws}$, delta′](self);
return(self).
```

```
$(c, \mathsf{delta})$ :
return(self).
```

```
$(c, \mathsf{delta}')$ :
return(self).
```

```
$(c_{\rm t}, \mathsf{delta}')$ :
self.$a'_{\rm ws}$ := false;
$y$ := copy[$\vec{a}$, $\vec{a}'$](self);
$y$ := copy[$\vec{a}'$, $\vec{a}''$](self);
$\underline{\text{Compute } \langle q, \gamma \rangle \text{ from } \vec{a},\ \vec{a}',\ \vec{a}''}$
    $\underline{\text{and assign the result to } \vec{a}}$;
$y$ := delta(self.$a_{\Rightarrow}$);
self.$a'_{\rm ws}$ := true;
return(self).
```

Figure A.4. Definitions of methods step and delta.

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

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

(C3′-1) The same as (C3-1);

(C3′-2) The same as (C3-2);

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

Lemma A.6: Suppose that $\mathbf{I} = (\nu, \mu)$ satisfies condition (C2) for some k ($k \geq 1$). Then, after the execution of step for o_1 under $\mathbf{I}$, 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 $l \in [1, r]$, $o_l.a_{\text{cont}}$ is true, i.e., $r = \max(\{0\} \cup \{j \mid \bigwedge_{l=1}^{j}(o_l.a_{\text{cont}} = o_l)\})$. Then, for each $i \in [1, k-r]$, $o_{r+i}.\vec{a}$ represents $I_r[i]$.

Proof: From the definition of methods step and delta, the value of $o_i.a_{\Rightarrow}$ ($i \in [1, k]$) 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 $i \in [1, k]$. Therefore, by the definitions of $(c_{\text{t}}, \mathsf{step})$, $(c_{\text{t}}, \mathsf{step}')$, and $(c_{\text{t}}, \mathsf{step}'')$, it is easily verified that delta is sequentially invoked on $o_1, \ldots, o_r$ 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 A.5 can be applied r times. Consequently, after the execution of step for o_1 under $\mathbf{I}$, $o_{r+i}.\vec{a}$ represents $I_r[i]$ for each $i \in [1, k-r]$. That is, (C4-3) holds.

Lastly, (C4-2) is satisfied because of (C3$'$-2) and the fourth line of $(c_{\text{t}}, \mathsf{step}'')$.
□

Lemma A.7: 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 ($i \in [1, k]$) such that $o_i.\vec{a}$ contains the accepting state q_{yes}, and $o_{k+1}.a_{\text{f}}$ otherwise.

Proof: By the first line of $(c_{\text{t}}, \mathsf{accept})$, $o_1.a'_{\text{yes}}$ 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 ($j \in [1, k]$). By the second line of $(c_{\text{t}}, \mathsf{accept}'')$, $o_j.a_{\text{yes}}$ is set to true (i.e., o_j) if $o_j.\vec{a}$ contains q_{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

```
$(c_{\mathrm{t}}, \mathsf{accept})$ :
$\mathsf{self}.a'_{\mathrm{yes}} := \mathsf{false}$;
$y := \mathsf{accept}'(\mathsf{self})$;
$\mathsf{return}(y)$.
```

```
$(c_{\mathrm{t}}, \mathsf{accept}')$ :
$y := \mathsf{if\_then}[a_{\mathrm{ws}}, \mathsf{accept}''](\mathsf{self})$;
$\mathsf{return}(y)$.
```

```
$(c_{\mathrm{t}}, \mathsf{accept}'')$ :
$\mathsf{self}.a_{\mathrm{ws}} := \mathsf{false}$;
$\underline{\mathsf{self}.a_{\mathrm{yes}} := \mathsf{true} \text{ if } q_{\mathrm{yes}} \text{ is stored in } \mathsf{self}.\vec{a}}$,
    $\underline{\text{and } \mathsf{self}.a_{\mathrm{yes}} := \mathsf{false} \text{ otherwise}}$;
$\underline{\text{Calculate } \mathsf{self}.a_{\mathrm{yes}} \vee \mathsf{self}.a'_{\mathrm{yes}} \text{ and}}$
    $\underline{\text{assign the result to } \mathsf{self}.a_{\mathrm{yes}}}$;
$y := \mathsf{copy}[a_{\mathrm{yes}}, a'_{\mathrm{yes}}](\mathsf{self})$;
$y := \mathsf{accept}'(\mathsf{self}.a_{\Rightarrow})$;
$\mathsf{return}(y)$.
```

```
$(c, \mathsf{accept})$ :
$\mathsf{return}(\mathsf{self})$.
```

```
$(c'_{\mathrm{t}}, \mathsf{accept}')$ :
$\mathsf{return}(\mathsf{self}.a'_{\mathrm{yes}})$.
```

```
$(c, \mathsf{accept}')$ :
$\mathsf{return}(\mathsf{self})$.
```

```
$(c, \mathsf{accept}'')$ :
$\mathsf{return}(\mathsf{self})$.
```

Figure A.5. Definition of method accept.

is some object o_i ($i \in [1, j]$) such that $o_i.\vec{a}$ contains q_{yes}, and $o_{j+1}.a'_{\mathrm{yes}}$ is set to false (i.e., $o_{j+1}.a_{\mathrm{f}}$) otherwise.

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

By Lemmas A.3–A.7 and the explanation in Section 3.2, the following lemma holds.

Lemma A.8: $\mathbf{S}_{M,x}$ is inconsistent if and only if M accepts x. □

Theorem 3.2 is obtained by Lemmas A.2 and A.8.