The VLDB Journal (1996) 5: 133–150

The VLDB Journal

Type-safe relaxing of schema consistency rules for flexible modelling in OODBMS

Eric Amiel[3], **Marie-Jo Bellosta**[2], **Eric Dujardin**[1], **Eric Simon**[1]

[1] INRIA Rocquencourt, projet RODIN, BP 105, F-78153 Le Chesnay Cedex, France; e-mail: lastname@rodin.inria.fr
[2] Lamsade, Université Paris Dauphine, F-75775 Paris Cedex 16, France; e-mail: bellosta@lamsade.dauphine.fr
[3] NatSoft, Air Center, CH-1214 Geneva, Switzerland

Edited by Matthias Jarke, Jorge Bocca, Carlo Zaniolo. Received September 15, 1994 / Accepted September 1, 1995

Abstract. Object-oriented databases enforce behavioral schema consistency rules to guarantee type safety, i.e., that no run-time type error can occur. When the schema must evolve, some schema updates may violate these rules. In order to maintain behavioral schema consistency, traditional solutions require significant changes to the types, the type hierarchy and the code of existing methods. Such operations are very expensive in a database context. To ease schema evolution, we propose to support exceptions to the behavioral consistency rules without sacrificing type safety. The basic idea is to detect unsafe statements in a method code at compile-time and check them at run-time. The run-time check is performed by a specific clause that is automatically inserted around unsafe statements. This check clause warns the programmer of the safety problem and lets him provide exception-handling code. Schema updates can therefore be performed with only minor changes to the code of methods.

Key words: Object-oriented databases – Schema evolution – Type safety – Covariance – Contravariance

1 Introduction

An object-oriented database schema contains the description of the types[1], type hierarchy, and methods used by all application programs. Types and method interfaces allow modelling of the complex objects coming from conceptual design, while method code and type representation define the implementation of objects. As a consequence, object-oriented databases must meet requirements arising from both a conceptual data modelling and a programming perspective.

From a programming point of view, it is highly desirable to guarantee type safety, for instance in order to protect the database against data corruption caused by type errors. To ensure type safety, object-oriented systems typically ensure that a schema satisfies three *behavioral consistency* rules. These rules are *sufficient* conditions that guarantee that no type error can occur during the execution of a method code. The *substitutability* rule says that if a type T_2 is a subtype of a type T_1 then whenever an instance of T_1 is expected in a variable assignment or a function invocation, it must be allowed to pass an instance of T_2. The *covariance* and *contravariance* rules impose constraints when a method is redefined for more specialized types. The covariance rule says that the return type must also be specialized. The contravariance rule says that the types of arguments that are not used for late binding must be more general. If a database schema satisfies these rules, it is said to be *behaviorally consistent*.

However, from a database modelling perspective, the schema must evolve in order to accommodate evolutions of the real world. As argued by Borgida (1988), this is particularly important in databases "where it is in general impossible or undesirable to anticipate all possible states of the world during schema design". The problem is that some schema updates may violate the behavioral consistency rules. For example, consider a database schema that contains a type $Patient$ having an attribute $doctor$ of type $Physician$. Suppose that we define a new type, called $Alcoholic$, as a subtype of $Patient$, and that the attribute $doctor$ inherited from $Patient$ is redefined to be of type $Psychologist$. Since a $Psychologist$ is (usually) not a $Physician$, the method that retrieves the $doctor$ attribute value of an alcoholic violates the covariance rule and the method that updates the $doctor$ attribute value of an alcoholic violates the contravariance rule.

There are also specific situations that are part of the (real-life) application that constitute violations of the behavioral consistency rules. For instance, in an hospital database, one may say that ambulatory patients are exactly like patients (i.e., $Ambulatory\_patient$ is a subtype of $Patient$) except that they have no hospital ward. This leads to the violation of the substitutability rule because the method that retrieves a $ward$ attribute value is not applicable to an instance of $Ambulatory\_patient$.

Existing systems have two attitudes with respect to this problem. One is to encourage the programmer to follow the rules, but not actually force him to do so (e.g., C++, or

Correspondence to: E. Simon

[1] We intentionally avoid talking about classes, which are viewed as types in some systems and as type extensions in others

O_2 for the contravariance rule). Inconsistent schemas are allowed and it is the programmer's responsibility to control what the program does and to avoid run-time type errors. The second attitude is to prevent the user from violating the rules. In this case there are several well-known solutions that lead to either changing the type hierarchy and introducing "fake" types, or breaking the type hierarchy and losing the advantages of polymorphism. These solutions may require significant changes to the code of methods. Both attitudes are clearly not satisfactory since they result in either unsafe code or substantial and artificial revisions to the schema.

The starting point of our research is that *exceptions* to the behavioral consistency rules should be supported to ease schema evolution and modelling. However, they should be checked at run-time to avoid type errors. In this paper, we propose to process every method source code and (1) determine whether a statement is unsafe, i.e., may result in a run-time type error, (2) automatically insert a "check" clause around every unsafe statement in the source code, and (3) let the user provide exception-handling code. The check clause is merely an if-then-else statement where the if-part performs a safety run-time check, the then-part contains the original statement, and the else-part contains the exception-handling code[2]. The insertion of check clauses warns the user about possible run-time type errors. The safety condition in the if-part of the "check" clause is expressed intensionally, thereby avoiding the reformulation of the condition when the schema changes. Our tool can also automatically generate some default exception-handling code. However, if the programmers provide their own exception-handling code then it has to be inspected by our tool.

Our proposed approach facilitates schema evolution by supporting exceptions, while guaranteeing that no run-time type error will occur. We focus on the motivations for such an approach and the type checking of statements in the presence of exceptions to behavioral consistency. Our results apply to object-oriented databases that support run-time method selection using either a single method's argument (mono-methods) or all method's arguments (multi-methods) as in recent systems like CLOS (Bobrow et al. 1988), Polyglot (Agrawal et al. 1991), and Cecil (Chambers 1992).

The paper is organized as follows. Section 2 introduces preliminary definitions about single- and multi-targeted methods, and defines the notion of consistent schema. Section 3 gives an overview of the problem, while Sect. 4 sketches the proposed solution. Section 5 introduces the material necessary to present our type system. Section 6 describes the type checking process allowing to distinguish between safe and unsafe statements. Section 7 describes how this process can be optimized. Section 8 establishes the relationships between the notions of consistency and safety. Section 9 relates our work with existing work, and Sect. 10 concludes the paper.

2 Schema consistency

In this section, we introduce our notations for the types and methods of a schema, mostly as defined by Agrawal et al. (1991). Then, we define the behavioral consistency rules and how they impact on structural consistency through encapsulation. Note that our notion of consistency is only concerned with typing, and not with semantics. It does not include issues such as integrity constraints [e.g., as in Formica and Missikoff (1994)] or business rules.

[2] We do not focus on the issue of designing specific language primitives for handling exceptions that can be harmoniously integrated with existing OO programming languages

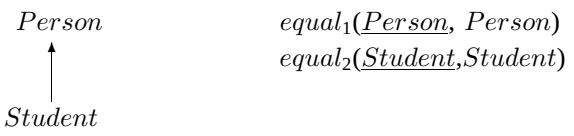


Fig. 1. A simple schema

2.1 Notations

We assume the existence of a partial ordering between types, called *subtyping* ordering, denoted by $\preceq$. Given two types T_1 and T_2, if $T_2 \preceq T_1$, we say that T_2 is a subtype of T_1 and T_1 is a supertype of T_2. As in other studies (Zdonik and Maier 1989; Bruce 1993; Danforth and Simon 1992), subtyping is a declared a relationship between types, which is decoupled from implementation decisions, and used solely to reflect operational similarities between different types.

To each *generic function* m corresponds a set of methods $m_k(T_k^1, \ldots, T_k^n) \rightarrow R_k$, where T_k^i is the type of the ith formal argument, and where R_k is the type of the result. We call the list of arguments $(T_k^1, \ldots, T_k^n)$ of method m_k the *signature* of m_k. An invocation of a generic function m is denoted $m(T_1, ..., T_n)$, where $(T_1, \ldots, T_n)$ is the signature of the invocation, and the T_is represent the types of the expressions passed as arguments. We shall use uppercase letters to denote type names, and lowercase letters to denote type instances, generic functions, methods, and method invocations.

In traditional object-oriented systems, functions have a specially designated argument, the *target*, whose run-time type is used to select the method to execute (method resolution). Multi-methods, first introduced in CommonLoops (Bobrow et al. 1986) and CLOS (Bobrow et al. 1988), provide a generalization of single-targeted methods by making all arguments targets. Multi-methods are now a key feature of several systems such as Polyglot (DeMichiel et al. 1993), Kea (Mugridge et al. 1991), Cecil (Chambers 1992), Dylan (Apple Computer 1994) and SQL3 (Melton 1994). Henceforth, we consider that methods are targeted on either one or all arguments. For the sake of uniformity, we shall assume that the p first arguments of a function (where p = 1 or p = n) are the target arguments. In the examples, we underline the target arguments in the signatures.

Example 2.1. Consider the type hierarchy of Fig. 1, and suppose we wish to define a generic function $equal$ for people and students. Since equality is defined differently for people and students, two methods $equal(\underline{Person}, Person)$ and $equal(\underline{Student}, Student)$ are needed to implement the generic function and we respectively denote them $equal_1$ and $equal_2$. Their signatures, given in Fig. 1, show that these methods have a single target argument. On invocation $equal$

Representation of type T	Update	Access
T = *tuple*$(\ldots, a_i : T_i, \ldots)$	$set\_a_i(\underline{T}, T_i)$	$a_i(\underline{T}) \rightarrow T_i$
T = *set*(T_1)	$insert\_element(\underline{T}, T_1)$ $remove\_element(\underline{T}, T_1)$	$empty?(\underline{T}) \rightarrow Boolean$
T = *list*(T_1)	$insert\_element(\underline{T}, T_1)$ $remove\_element(\underline{T}, T_1)$	$empty?(\underline{T}) \rightarrow Boolean$ $retrieve\_element\_at(\underline{T}, Integer) \rightarrow T_1$

Fig. 2. Signatures of representation methods

$Person : tuple(name : String,$
$\quad onBankAccount : Float,$
$\quad inLifeInsurance : Float,$
$\quad resources : List(Resources))$

$Student : tuple(name : String,$
$\quad onBankAccount : Float,$
$\quad inLifeInsurance : Float,$
$\quad resources : List(Resources),$
$\quad cardID : String)$

Fig. 3. Representation of types

($Person$, $Student$), the run-time method dispatcher will select method $equal_1$ based on the first target argument.

Given a generic function invocation, the selection of the corresponding method follows a two-step process: first, based on the types of the target arguments, a set of applicable methods is found and, second, a precedence relationship between applicable methods is used to select what is called the *Most Specific Applicable* (MSA) method. Intuitively, a precedence relationship determines which applicable method most closely matches a function invocation. Given a signature $s = (T_1, \ldots, T_n)$ and a function invocation $m(s)$, if m_i and m_j are applicable to $m(s)$ and, according to a particular method precedence ordering, m_i is more specific than m_j for s, noted $m_i <_s m_j$, then m_i is a closer match for the invocation. When the method precedence ordering does not depend on signatures, i.e., $\forall s\ m_i <_s m_j$, we just write $m_i < m_j$.

In the remainder of this paper, we assume that for any function invocation $m(T_1, \ldots, T_n)$, if there is an applicable method, then there always exists an MSA method and this method is unique. We call this the unique most specific applicable (UMSA) property. Agrawal et al. (1991) examine different possible method precedence orderings and focus on global type precedence and inheritance order precedence, which enforce the UMSA property in case of multiple inheritance and multiple targets. However, we insist that our results do not depend on the means by which the UMSA property is enforced.

Types can be represented using different data structures such as set, tuple and list. We assume that the system enforces the *encapsulation* of the representation of types. Each type has a set of built-in *representation* operations that enable to manipulate (i.e., access and update) the state of instances of that type. For our purpose, we consider a subset of the operations defined in the ODMG object model (Cattell 1994). *Representation* methods perform built-in operations on each type. The table in Fig. 2 summarizes the signatures of their representation methods. Moreover, it is possible to iterate over the elements of a collection, i.e. a set or a list, by using a $foreach$ statement.

Example 2.2. As shown in Fig. 3, both a $Person$ and a $Student$ have several income resources used to compute taxes. The total amount of their financial resources is also divided between a bank account and a life insurance. Additionally, $Students$ have a *cardID*. The following invocation allows the insertion of a new resource r in the resources list of a person p: $insert\_element(resources(p), r)$.

2.2 Behavioral consistency rules

Object-oriented typing theory defines three consistency rules to guarantee that no type error can occur during the execution of a method code. The first two rules impose constraints on the types returned by methods and the types of methods arguments. The third rule relaxes the condition of type equality on substitution operations (variable assignment or parameter passing) to take into account the subtyping relationship. The three rules are:

Covariance rule. Given two methods $m_i(T_i^1, \ldots, T_i^n) \rightarrow R_i$ and $m_j(T_j^1, \ldots, T_j^n) \rightarrow R_j$, where, for some signature s, $m_i <_s m_j$, then $R_i \preceq R_j$.

Contravariance rule. Given two single-targeted methods (p = 1), $m_i(T_i^1, \ldots, T_i^n) \rightarrow R_i$ and $m_j(T_j^1, \ldots, T_j^n) \rightarrow R_j$, where, for some signature s, $m_i <_s m_j$, then $\forall k \geq 2$, $T_j^k \preceq T_i^k$.

Substitutability rule. Given two types T_1 and T_2, an instance of T_2 can be substituted to an instance of T_1 if and only if $T_2 \preceq T_1$ (*substitutability condition*).

The covariance rule is called consistency by Agrawal et al. (1991). The contravariance rule was originally developed for subtyping of functions (Cardelli 1984), and has been extended to subtyping on partially targeted methods (Mc Menzie 1992; Danforth 1990). The substitutability rule is the basis of inclusion polymorphism (Cardelli and Wegner 1985).

2.3 Structural consistency rules

As shown by Kemper and Moerkotte (1994), the behavioral consistency rules on representation methods imply structural consistency rules on the representation because of encapsulation. These rules state that the representation of the supertypes must be included in the representation of their sub-

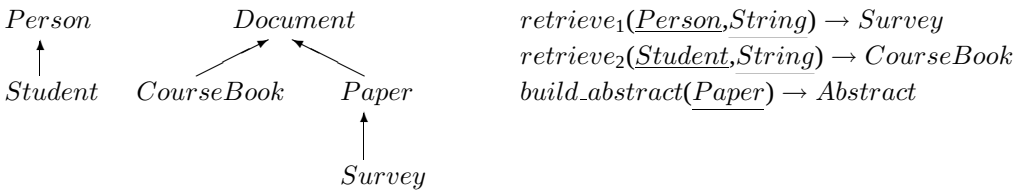


Fig. 4. Document hierarchy

types, and disallow the redefinition of attribute types of tuples and element types of collections. Let T_i and T_j be two types, such that $T_i \preceq T_j$, we have:

Tuple subtyping rule. If the representation of T_j is $tuple(a_j^1 : T_j^1, \ldots, a_j^{n_j} : T_j^{n_j})$, then the representation of T_i is $tuple(a_i^1 : T_i^1, \ldots, a_i^{n_i} : T_i^{n_i})$, with $\{a_j^1, \ldots, a_j^{n_j}\} \subseteq \{a_i^1, \ldots, a_i^{n_i}\}$, and for all $k_j \leq n_j$, all $k_i \leq n_i$, $a_j^{k_j} = a_i^{k_i} \Rightarrow T_j^{k_j} = T_i^{k_i}$.

Set subtyping rule. If the representation of T_j is $set(T_j^1)$, then the representation of T_i is $set(T_i^1)$, with $T_i^1 = T_j^1$.

List subtyping rule. If the representation of T_j is $list(T_j^1)$, then the representation of T_i is $list(T_i^1)$, with $T_i^1 = T_j^1$.

These rules restrict the rules of structural subtyping defined by Cardelli and Wegner (1985), that also appear in the work of Baneerje et al. (1987). The rules of Baneerje et al. (1987) state that a tuple-structured type T_2 is a subtype of T_1 iff T_2 has all the attributes of T_1, and if the types of common attributes in T_2 are subtypes of those in T_1. Thus, representation methods available on T_1 instances are also available on T_2 instances. However, as noted by Kemper et al. (1994), the update operations do not respect the contravariance rule. Zdonik and Maier (1989) generalizes this problem to the redefinition of method parameters with subtypes, called *specialization via constraints*. They show that specialization via constraints leads to run-time type errors that cannot be handled by type checking at compile-time.

3 Problem overview

In this section, we first define exceptions to behavioral consistency and give several examples of each kind of exception. Next, we relate the violations of structural consistency rules to behavioral exceptions. We then summarize the type errors possibly induced by these exceptions. Finally, we present solutions recommended by object-oriented design methods to avoid exception to consistency.

3.1 Exceptions to behavioral consistency

We define a *behavioral exception* as the violation of one of the three behavioral consistency rules. The non-respect of the covariance rule yields *return-exceptions*, while the non-respect of the contravariance rule yields *argument-exceptions*. Violations of the substitutability rule yields two kinds of exceptions. The first one is when a signature is disallowed for a generic function, although the substitutability condition for parameter passing is satisfied. The second one is when the substitutability condition is violated during assignment or parameter passing. These exceptions are called *disallowed signature* and *illegal substitution*, respectively.

In the following, we only consider return-exceptions, argument-exceptions, and disallowed signatures as possible exceptions to the behavioral consistency rules. Indeed, illegal substitutions have more far-reaching consequences on static type checking than the three other kinds of exceptions.

3.1.1 Return-exceptions

Method m_i is a return-exception to method m_j iff $m_i <_s m_j$ for some signature s, and the return type of m_i is not a subtype of the return type of m_j.

Imposing covariance on the result ensures that whatever method is selected at run-time, its result is a subtype of the type expected by the context of the invocation.

Example 3.1. Consider the schema in Fig. 4, which respects the structural consistency rules. Consider the generic function $retrieve$ that searches a document database according to the profile of the library user and his topic of interest. A person receives a survey, while a student is presented the course book relevant to his level. Thus, the method $retrieve_2$ is a return-exception to $retrieve_1$, capable of yielding type errors. For instance, suppose that a generic function $build\_abstract$ uses pattern matching to extract the abstract of a paper and course books have no abstract. The statement $build\_abstract(retrieve(aPerson,$"$database\ systems$"$))$ leads to a run-time error if $aPerson$ refers to a student at run-time, as there is no applicable $build\_abstract$ method.

Return-exceptions can also cause illegal substitutions which can then lead to run-time type errors.

Example 3.2. Consider the following assignment of a variable $mySurvey$ of type $Survey$: $mySurvey \leftarrow retrieve(myPerson,$ "$database\ systems$"$)$. If $myPerson$ refers to a student at run-time, a course book is assigned to $mySurvey$, which constitutes an illegal substitution. The invocation $build\_abstract(\ mySurvey)$ has no applicable method, thereby causing a run-time type error.

3.1.2 Argument-exceptions

Method m_i is an argument-exception to method m_j iff $m_i <_s m_j$ for some signature s, and there exists a non-target argument T_i^k of m_i which is not a supertype of T_j^k.

Argument-exceptions only occur in systems with single-targeted functions where run-time method selection does

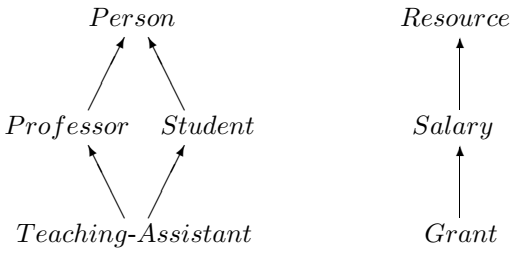


Fig. 5. Disallowed signature

not check that the non-target arguments of an invocation are subtypes of the non-target formal arguments of the selected method. This may result in illegal substitutions when the actual arguments are assigned to the formal arguments. However, the possibility of specializing any argument of a method is clearly needed in practice, and for this reason, most object oriented systems do not actually enforce the contravariance constraint [see Cattaneo et al. (1993), Meyer (1992), Connor and Morrison (1992) and O_2 Technology (1992)].

Example 3.3. Consider the schema of Fig. 1 where $Student$ is a subtype of $Person$. The invocation $equal(myPerson_1, myPerson_2)$ leads to the selection of $equal_2$ if the target argument, $myPerson_1$, refers to a student at run-time. But if the type of $myPerson_2$ refers to a person, an illegal substitution occurs between the formal argument of type $Student$ and $myPerson_2$. Then, in the body of $equal_2$, applying on this argument a function that is only defined for $Student$ (e.g., to access the $cardID$ attribute) causes a run-time error as there is no applicable method. Note that the representation of types $Student$ and $Person$ conform to the structural subtyping rules.

3.1.3 Disallowed signatures

Signature s is a disallowed signature of m iff invoking m on s is forbidden, although an MSA method for $m(s)$ existss.

Example 3.3 shows that some signatures should be disallowed because they imply illegal substitutions between non-target actual and formal arguments. We refer to these signatures as *implicitly* disallowed signatures, as they can be inferred from argument-exceptions. However, some disallowed signatures cannot be inferred and must be explicitly given by the user as part of the semantics of the application. We call these signatures *explicitly* disallowed signatures. Following Borgida (1988), they are defined as excuses on the generic function: $excuse\ m\ on\ s_1,...,s_x$.

Example 3.4. Consider the schema of Fig. 5 where $Professor$ and $Teaching\text{-}Assistant$ have the same structure as $Person$ with an additional attribute $Dept$. Moreover, $Teaching\text{-}Assistant$ also has a $cardID$ attribute, like $Student$. Suppose we update the schema by adding a function $allocate$ that updates the $resources$ attribute and manages the financial resources by distributing money between the bank account and the life insurance, depending on a complex criterion. This function has two methods $allocate_1(\underline{Person}, Resource)$ and $allocate_2(\underline{Professor}, Salary)$. A professor receives a salary and some grants are allocated for his research projects. A student can also receive a salary and/or a grant. A teaching-assistant can only receive a salary. Thus, the method $allocate_2$ is not applicable to signature ($Teaching\text{-}Assistant$, $Grant$), which is disallowed. Finally, the specialization of the second argument induces two implicitly disallowed signatures ($Professor$,$Resource$) and ($Teaching\text{-}Assistant$,$Resource$). All other signatures are allowed.

3.2 Exceptions to structural consistency

Because of encapsulation, exceptions to structural consistency entail exceptions to the behavioral consistency rules. There are two kinds of exceptions to structural consistency: *data structure mismatch* and *component type redefinition*. A data structure mismatch arises in two cases: (1) when different data structures are used to build the representation, and (2) in the case of *inapplicable attributes*, i.e., attributes of the supertype that do not appear in the subtype (see, e.g., Borgida (1988)). A component type redefinition arises when a subtype has the same data structure as its supertype. This redefinition focuses on the types of tuples' attributes and collections' elements.

3.2.1 Data structure mismatch

In the case of a data structure mismatch, some or all of the representation methods of the supertype cannot be applied to objects of the subtypes. This corresponds to explicitly disallowed signatures. Figure 6 summarizes these disallowed signatures in the case of different data structures between a type T_1 and its subtype T_2. Finally, disallowed signatures due to an inapplicable attribute $a_i : T_1^i$ of a type T_1 with respect to its subtype T_2 are (T_2) and (T_2, T_1^i) for a_i and set_a_i, respectively.

Example 3.5. Consider the schema of Fig. 7, borrowed from Danforth and Simon (1992). A data structure mismatch occurs between $Polygon$ and $Square$, because a square is obviously a kind of polygon, but the data structure of these types differ. Hence the representation methods of $Polygon$ are not applicable to squares.

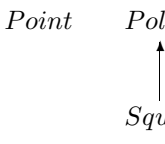
Example 3.6. Suppose we are given a schema where $Ambulatory\_Patient \preceq Patient$ and we want to update the schema by adding an attribute $ward$ for $Patient$. This attribute is irrelevant to subtype $Ambulatory\_Patient$. Thus, accessing or updating the ward of an $Ambulatory\_Patient$ should not be allowed. Then, ($Ambulatory\_Patient$) and ($Ambulatory\_Patient$, $Ward$) are disallowed signatures for methods $ward(Patient) \rightarrow Ward$ and $set\_ward(Patient, Ward) \rightarrow Ward$, respectively.

3.2.2 Component type redefinition

As shown in by Kemper and Moerkotte (1994), Cook (1989), Connor et al. (1991), Danforth and Simon (1992), a component type redefinition between a type T_1 and its subtype T_2 leads to one of the following exceptions:

Representation of supertype T_1	built-in methods	explicitly disallowed signatures
T_1 = $tuple(\ldots, a_i : T_1^i, \ldots)$	set_a_i	(T_2, T_1^i)
	a_i	(T_2)
T_1 = $set(T)$	$insert\_element$	(T_2, T)
	$remove\_element$	(T_2, T)
	$empty?$	(T_2)
T_1 = $list(T)$	$insert\_element$	(T_2, T)
	$remove\_element$	(T_2, T)
	$empty?$	(T_2)
	$retrieve\_element\_at$	$(T_2, Integer)$

Fig. 6. Explicitly disallowed signatures for tuples, sets and lists



$Point$ = $tuple(x : Integer, y : Integer)$
$Polygon$ = $list(Point)$
$Square$ = $tuple($ $upper\_left\_corner : Point,$
$side\_length : Integer)$

Fig. 7. Data structure mismatch

A return-exception of the access methods if the type appearing in T_2 is not a subtype of the corresponding type in T_1.

An argument-exception of the update methods if the type appearing in T_2 is not a supertype of the corresponding type in T_1.

Example 3.7. Suppose that $Patient$ is a type with an attribute $doctor$ of type $Physician$. Suppose we want to add a new type $Alcoholic$ to the schema as a subtype of $Patient$, where attribute $doctor$ is of type $Psychologist$. The updated schema is shown on Fig. 8. As $Psychologist$ is not a subtype of $Physician$, the method $doctor_2$ is a return-exception to method $doctor_1$. This exception can cause type errors as shown below. Consider the method $refund(\underline{Hospital}, \underline{Dollar})$ that refunds the expenses of a patient to the hospital he was treated in, and the function $refunding$ that refunds a set of patients using method $refund$.

```
refunding(patients : PatientSet)
{ foreach p in patients do
    refund(hospital(doctor(p)),bill(p));
 end do; }
```

As psychologists are not affiliated to a hospital, unlike physicians, the invocation $hospital$ ($doctor($ $myPatient$)) causes an error if $myPatient$ refers to an alcoholic at run-time as there is no applicable method for invocation $hospital(Psychologist)$.

Example 3.8. Consider the types in Fig. 9. The two representation methods $insert\_element_1(\underline{PersonList}, Person)$ and $insert\_element_2(\underline{StudentList}, Student)$ constitute an argument-exception.

3.2.3 Structural consistency and behavioral consistency

Figure 10 summarizes the relationships between the two kinds of exceptions to structural consistency and the three kinds of exceptions to behavioral consistency. An arrow from the structural exception x to the behavioral exception y means that x leads to y. For the remainder of the paper, we only consider exceptions to behavioral consistency, as they also capture exceptions to structural consistency. We define a database schema to be *consistent* iff every method satisfies the behavioral consistency rules.

3.3 Exceptions to consistency and type safety

A program is type safe if, during the execution of every statement, no error can occur due to the absence of an MSA method for invocation. The purpose of static type checking is to verify at compile-time that a program is type safe. To this end, for each statement of a method code, the declared types are used to check that (1) every invocation has an MSA method and (2) no illegal substitution may occur. If the above two conditions are satisfied, a statement is correct; otherwise, it is incorrect and there is a type error.

The central problem introduced by exceptions to behavioral consistency is that a correct statement may be unsafe, i.e., yield a type error at run-time. Thus, in presence of exceptions to behavioral consistency, type checking must further partition correct statements into *safe* and *unsafe* statements.

Figure 11 summarizes the relationships between the three different kinds of exceptions to behavioral consistency (bottom of Figure) and the three kinds of type errors at run-time (top of Figure); an arrow from x to y means that an exception of kind x may lead to a type error of kind y at run-time.

3.4 Solutions to avoid exceptions to consistency

Object-oriented design offers several solutions to the problems of consistency set by some schema updates. They modify the type hierarchy and the code of methods or introduce new methods. These solutions avoid return-exceptions and explicitly disallowed signatures, but not argument-exceptions. However, they involve important modifications of the type hierarchy or the code of methods. In a database context, this

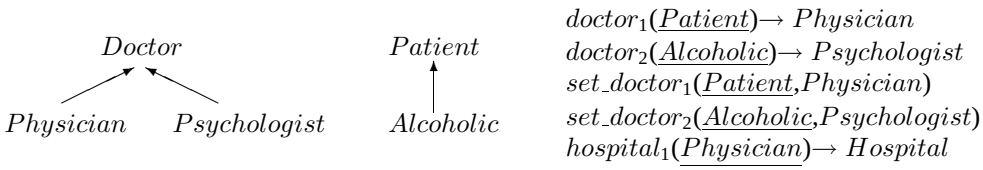


Fig. 8. Doctor and patient hierarchy

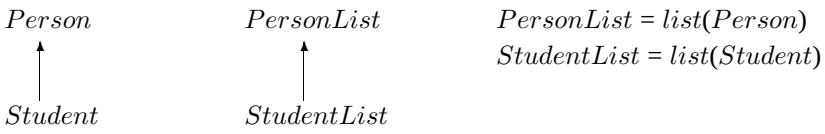


Fig. 9. Argument-exception in component type redefinition

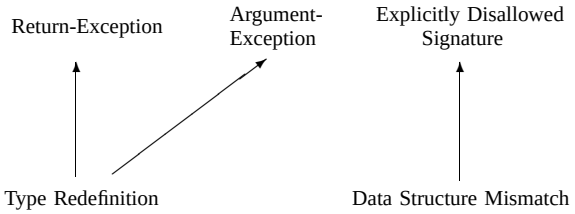


Fig. 10. Subtyping rules violations and exceptions

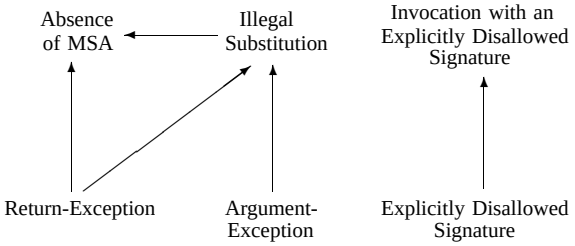


Fig. 11. Exceptions to consistency and type errors

can be expensive since changes to the types must be propagated to the persistent instances. Most importantly, the burden of implementing these solutions is left to the programmer. We examine four of these solutions on Example 3.7.

The first solution eludes the problem by renouncing to make $Alcoholic$ a subtype of $Patient$. Thus, the advantages of polymorphism are lost; alcoholics and patients must be stored in different sets and they must be handled separately by different methods, despite their similarities.

The second solution retains the advantages of polymorphism for the methods that use only the similarities between $Alcoholic$ and $Patient$. This solution involves a new intermediate type to represent the common part, in our case $Patient$ without attribute $doctor$. This can be achieved in two ways, illustrated in Fig. 12: (1) modify $Patient$ by removing attribute $doctor$ and create a subtype $Patient\_treated\_by\_Physician$, or (2) create $Patient0$ as a supertype of $Patient$, to represent patient without attribute $doctor$. In both cases, $Alcoholic$ is made a subtype of the intermediate type. In methods that do not use the difference between alcoholics and regular patients and that do not call methods using this difference, patients and alcoholics can be manipulated as being of the intermediate type.

The first problem with this solution is the multiplication of artificial intermediate types, like $Patient0$, which is combinatorial in nature [see Borgida (1988)] as they represent objects with a subset of the attributes of $Patient$. The second problem is that retaining polymorphism through the use of an intermediate type only works for some methods. In our previous example, every method that calls $refunding$ cannot pass a heterogeneous set containing both regular patients and alcoholics. This is a major disadvantage in a database context, where applications are collection-oriented. In this case, solution (2) is preferable because it only requires modification of methods but not existing instances.

The third solution involves re-conciliating physicians and psychologists by declaring a method $hospital$ on $Doctor$. This method is defined as simply returning a NULL reference to indicate that doctors who are psychologists are not affiliated to hospitals. In this way, invocation $hospital(doctor(p))$ is not an error even if p refers to an alcoholic at run-time. The problem with this solution is the definition of artificial methods, like $hospital(\underline{Doctor})$, which seems to indicate that a function is available on a certain type while it is actually not. Moreover, it is the responsibility of the programmer to know that $hospital$ invoked with a doctor may return a NULL reference and that the result of the function must be tested. In our example, $refunding$ must be rewritten as:

```
refunding(patients:PatientSet)
{ foreach p in patients do
         if hospital(doctor(p)) <> NULL
            refund(hospital(doctor(p)),
                   bill(p));
 end do; }
```

A last solution involves defining two intermediate methods $foo(\underline{Patient})$ and $foo(\underline{Alcoholics})$. The first encapsulates the original statement of refunding the hospital, the second defines what must be done in the case of an alcoholic. Method $refunding$ is then rewritten to call foo on patients:

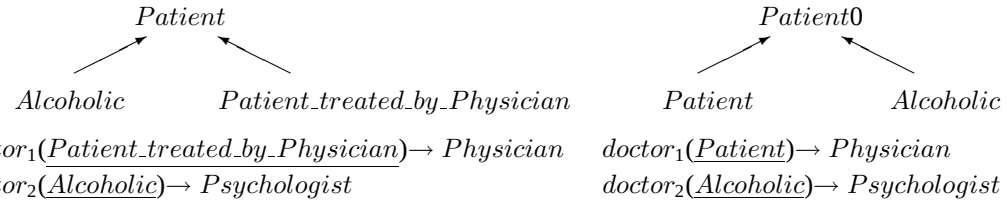


Fig. 12. Intermediate supertype creation

```
refunding(patients : PatientSet)
{ foreach p in patients do
         foo(p);
 end do; }

foo(p:Patient)
{ refund(hospital(doctor(p)),bill(p));}

foo(p:Alcoholic)
{/* handles the case of alcoholics */}
```

The problem with this solution is the multiplication of artificial switching methods.

In conclusion, painful aspects of these solutions are either the creation of new intermediate types, the addition of new artificial methods, the renunciation of polymorphism by not declaring a type as a subtype of another one, or the intervention of the programmer to test the result of methods that may return NULL values. These modifications are costly in a schema evolution context. Furthermore, they are defined by the user on an ad hoc basis.

4 The proposed solution

Our solution aims at allowing subtyping with exceptions to consistency, while enforcing type safety. In this section, we introduce the *check statements*, that allow acceptance of unsafe statements due to exceptions while guaranteeing that no type error can occur at run-time. We then show the impact of schema evolution on these check statements. We finally sketch the steps of the type-checking process.

4.1 Check statements

Check statements embed every statement identified as unsafe at compile-time – as shown in Fig. 13. The condition part checks that the unsafe statement is correct at run-time, and if it is, the statement is executed. Otherwise, an exception-handling code is executed. Check statements enable the user to be warned about the possibility of run-time failure, let the user provide exception handling code, and perform dynamic type checking of the unsafe statement.

Throughout this paper, we consider statements that are either function invocations or variable assignments, as shown in Fig. 14. Dynamic type checking involves evaluating their arguments, which may be invocations of functions. Verifying the correctness leads to execution of these functions twice, in the condition and unsafe statement parts. In case of functions with side-effects, the second execution is undesirable. To overcome this limitation, subexpressions of the

```
CHECK <condition>
      <unsafe statement>
ELSE
      <exception-handling code>
END
```

Fig. 13. Check statements

$$
\begin{aligned}
\mathit{statement} &::= \mathit{assignment} \mid \mathit{invocation} \\
\mathit{assignment} &::= \mathit{variable} \leftarrow \mathit{expression} \\
\mathit{invocation} &::= \mathit{function\_name}(\mathit{expression}^*) \\
\mathit{expression} &::= \mathit{variable} \mid \mathit{constant} \mid \mathit{invocation}
\end{aligned}
$$

Fig. 14. Pseudo-EBNF grammar of statements

arguments can be bound to variables local to the CHECK. These variables are untyped [as "void" variables in Kemper and Moerkotte (1994) or "dynamics" in Abadi et al. (1989)], and can be used both in the unsafe statement part of the CHECK and its exceptional-handling code instead of the original invocations with side-effects, so that these invocations are not evaluated twice[3].

The condition part is *intensionally* mentioned, in the sense that the types for which the exception occurs are not explicitly given. Evaluating the correctness condition involves taking the run-time type of the expressions composing the statement and verifying that the statement is correct with these types, which amounts to query the schema at run-time. Depending on the statement, two expressions of the condition are defined, as shown in Fig. 15.

Example 4.1. In Example 3.1 invocation $build\_abstract(retrieve(myPerson, \text{“}database\ systems\text{”}))$ is unsafe because $myPerson$ may contain a student. Thus, this statement must be surrounded by a CHECK. Let us assume that the generic function $retrieve$ has a side-effect, e.g., it increments a counter of users. In order to prevent the increment from happening twice, we shall use a foo variable that stores the result of $retrieve$ in the CHECK condition. Then, variable foo is used in both the unsafe statement and the exception-handling code, as shown in Fig. 16.

Some schema evolution operations require to re-evaluate existing programs, which possibly leads to add or delete CHECK statements. Additions are due to newly unsafe statements, and deletions are due to previously unsafe statements becoming safe. The intensional form saves one from reformulating existing CHECK statements retained by the new evaluation.

Example 4.2. In Example 3.7, suppose that a new type of physician, $FamilyPractitioner$, is introduced, for which

[3] In the following sections, we assume that only functions without side-effects are used as the invocation's arguments of unsafe statements

Unsafe Statement	Condition Part
Invocation $m(e_1, \ldots, e_n)$	m `IS CORRECT ON` $(e_1, \ldots, e_n)$
Assignment $v \leftarrow e$	e `MAY BE ASSIGNED TO` v

Fig. 15. Expression of CHECK conditions

```
CHECK build_abstract IS CORRECT
     ON foo:=retrieve(myPerson,"database systems")
   build_abstract(foo) ;
ELSE
   introduction(foo) ;
END
```

Fig. 16. CHECK of an invocation

$hospital$ is not applicable (i.e., an explicitly disallowed signature). As our correction test is intensional, the check does not need to be reformulated as shown in Fig. 17.

Example 4.3. Suppose that a new type of patient, $Tubercular$, is introduced, whose expenses are expressed in Swiss francs (SF). As $bill(p)$ may return Swiss francs, and hospitals may only be refunded Dollars, there exists a signature $(Hospital, SF)$ for which no $refund$ method is applicable. Thus $refund(hospital(doctor(p)), bill(p))$ is unsafe, even when $hospital(doctor(p))$ is safe. As shown in Fig. 18, a nested check statement must be generated.

4.2 Type checking process

For every statement, the proposed type checking process works as follows:

1. Determine whether the statement is incorrect, unsafe or safe.
2. If the statement is incorrect, report the type error.
3. If the statement is unsafe, generate the appropriate check statements.
4. Prompt the user for exception-handling code.
5. Type check the statements of the exception-handling code.

In the first step, determining if a statement is correct uses the types known at compile-time, while determining if it is safe relies on the potential types at run-time. In the third step, the generation of the check statement must consider that several subexpressions of a statement may be unsafe. In such cases, check statements must be nested. The main problem with nested checks is to avoid unnecessary checks: indeed, when unsafe subexpressions share some variables or some subexpressions, checks may become redundant. The basic idea to minimize the number of checks is to have the type checker infer the possible run-time types of subexpressions along a chain of nested checks (equivalent to a chain of conditionals). The fourth step is deferred until the whole program has been type-checked, so that the user can give, at the same time, the exception-handling code for all unsafe statements. In the fifth step, the types inferred along the checks are used to type-check the exception-handling code in place of the types known at compile time. Because

```
CHECK hospital IS CORRECT ON (doctor(p))
      refund(hospital(doctor(p)), bill(p))
ELSE
      /* exceptional statement to be provided
         by the user */
END
```

Fig. 17. Schema evolution without generation of a new Check

```
CHECK hospital IS CORRECT ON (doctor(p))
   CHECK refund IS CORRECT ON (hospital(doctor(p)), bill(p))
     refund(hospital(doctor(p)), bill(p))
   ELSE
     /* user-provided exceptional statement */
     END
ELSE
   /* user-provided exceptional statement */
END
```

Fig. 18. Schema evolution with generation of a new check

of space limitations, we only describe the first step of this process.

5 Basic definitions

In this section, we introduce the notions of method applicability, exact type, cover of a signature, and range and disallowed signature of a method.

Total match and target match. Let $m_k(T_k^1, \ldots, T_k^n)$ and $m(T_1, \ldots, T_n)$ be a method and a function invocation, respectively for a generic function m. Then, m_k is said to be a *total match* for the invocation iff $\forall i \in \{1, \ldots, n\}$, $T_i \preceq T_k^i$, and m_k is said to be a *target match* for the invocation iff $\forall i \in \{1, \ldots, p\}$, $T_i \preceq T_k^i$ (p is the number of target arguments).

By extension, we talk about a method as being a total or target match for a signature. Note that in multi-targeted systems, the two notions merge, i.e., every target match is a total match.

Method applicability. A method $m_k(T_k^1, \ldots, T_k^n)$ is *applicable* to a function invocation $m(T_1, \ldots, T_n)$ if m_k is a target match for the invocation.

Consider again Fig. 1 and suppose that $equal$ is invoked with $equal(Student, Person)$. Both methods $equal_1$ and $equal_2$ are applicable because they are both target match to this invocation. However, $equal_1(\underline{Person}, Person)$ is a total match for the invocation and $equal_2(\underline{Student}, Student)$ is not a total match.

In the following, we use a function MSA which, given an invocation $m(T_1, \ldots, T_n)$, returns the MSA method m_k for this invocation – if any – and a specific method "$m_\top$" otherwise. The method $m_\top$ uses a specific "impossible" type, noted $T_\top$, as the type of its arguments and result. $T_\top$ is in strict supertype relation with all other types, i.e., $\forall T,\ T \prec T_\top$. This special method is defined for every generic function. MSA is used at run-time as the method dispatcher.

We now introduce the notion of *exact type* of an expression. The type of a constant c declared of type T is *exactly* T and not any type $T' \preceq T$. Similarly, the object resulting from an explicit "new" creation instruction is exactly

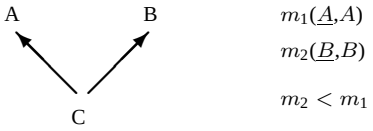


Fig. 19. Example schema

the type given as argument to "new". Thus, a variable that gets assigned the result of a "new" instruction is also of an exact type. Exact typing applies to expressions that appear as actual arguments of invocations or as right-hand side of assignments.

Exact typing. At compile-time, an expression e is said to be of an *exact type* T, denoted $e : \overline{T}$, iff any object referenced by e at run-time is of type T and not of any type T' such that $T' \prec T$.

Note that, by default, any expression e is of *free type* T, denoted $e : T$, i.e., e may yield at run-time an object of any type $T' \preceq T$. We shall use letter τ to indifferently refer to $\overline{T}$ and T when typing an expression.

Signature of expressions. The signature of a tuple of expressions $(e_1 : \tau_1, \ldots, e_n : \tau_n)$ is the tuple $(\tau_1, \ldots, \tau_n)$. The signature of a method $m_k(T_k^1, \ldots, T_k^n) \rightarrow R_k$ is the signature of its formal arguments, i.e., $(T_k^1, \ldots, T_k^n)$. The signature of an invocation $m(e_1, \ldots, e_n)$ with $e_1 : \tau_1, \ldots, e_n : \tau_n$ is the signature of its actual arguments, i.e., $(\tau_1, \ldots, \tau_n)$. Abusively, we shall call *signature* any tuple of free or exact types $(\tau_1, \ldots, \tau_n)$, and omit their associated expressions.

Cover of a signature. Let s be a signature $(\tau_1, ..., \tau_n)$. The *cover* of s, denoted by $cover(s)$ is defined as:

$$cover(s) = \{(U_1, \ldots, U_n) \mid \forall i \in \{1, \ldots, n\} \begin{cases} U_i \preceq T_i & \text{if } \tau_i = \underline{T_i} \ (\tau_i \text{ is free}) \\ U_i = T_i & \text{if } \tau_i = \overline{T_i} \ (\tau_i \text{ is exact}) \end{cases}$$

By extension, we also define the cover of a method m_i as the cover of its signature. Note that $cover(m_i)$ is the set of signatures for which m_i is a total match.

Example 5.1. Using the type hierarchy in Fig. 19, we have:

$cover(A, \overline{A}) = \{(A, A), (C, A)\}$

$cover(m_1) = cover(A, A) = \{(A, A), (A, C), (C, A), (C, C)\}$

$cover(m_2) = cover(B, B) = \{(B, B), (B, C), (C, B), (C, C)\}$

Well-typed signatures. The well-typed signatures of a generic function m, denoted $well\text{-}typed(m)$, is the union of the covers of all the methods associated with m:

$$well\text{-}typed(m) = \bigcup_{m_i} cover(m_i)$$

Example 5.2. Considering the example of Fig. 19, we have:

$well\text{-}typed(m)$

$= \{(A, A), (A, C), (C, A), (C, C), (C, B), (B, B), (B, C)\}$

Intuitively, $well\text{-}typed(m)$ represents the set of the invocation's signatures of m for which there exists a method m_i that is a total match.

Range of a method. Let m_i be a method for a generic function of arity n, and m a function invocation. The *range* of m_i, noted $range(m_i)$, is the set of signatures for which m_i is the MSA method:

$$\text{range}(m_i) = \{(T_1, \ldots, T_n) \in \text{well-typed}(m) | \text{MSA}(m(T_1, \ldots, T_n)) = m_i\}$$

Example 5.3. Considering the example of Fig. 19, we have:

$range(m_1) = \{(A, A), (A, C)\}$

$range(m_2) = \{(B, B), (B, C), (C, A), (C, C), (C, B)\}$

As the applicability of a method relies on a target match, we take for m_1 (or m_2), all signatures (T, T') in $well\text{-}typed(m)$ such that $T \preceq A$ (or $T \preceq B$). Observe that for signatures (C, A), (C, C), and (C, B), m_1 and m_2 are both applicable but since $m_2 < m_1$, these signatures belong to the range of m_2. Finally, note that $(C, A) \in range(m_2)$ but $(C, A) \notin cover(m_2)$. This is a consequence of single-targeting.

Explicitly disallowed signatures of a method. The set of explicitly disallowed signatures of a method m_i, noted $explicit(m_i)$, is the set of explicitly disallowed signatures of m that belong to the range of m_i.

These signatures are both in the range *and* the cover of m_i, as they correspond to the user's wish to forbid some otherwise type correct invocations. Thus,

$$explicit(m_i) \subseteq cover(m_i).$$

Example 5.4. Let us reconsider the schema introduced in Example 3.4, but for brevity, let the types be P for $Person$, Pr for $Professor$, S for $Student$, TA for $Teaching-Assistant$, R for $Resource$, Sa for $Salary$ and G for $Grant$. Consider the method $allocate_2$. We have: $range(allocate_2) = \{(Pr, R), (Pr, SA), (Pr, G), (S, R), (S, Sa), (S, G), (T, R), (T, Sa), (T, G)\}$ and $cover(allocate_2) = \{(Pr, Sa), (Pr, G), (T, Sa), (T, G)\}$.
Finally, we can see that $explicit(allocate_2) = \{(T, G)\}$ is included in $cover(allocate_2)$.

Implicitly disallowed signatures of a method. The set of implicitly disallowed signatures of a method m_i, noted $implicit(m_i)$ is given by:

$$implicit(m_i) = \{(T_1, \ldots, T_n) \in range(m_i) | T_{p+1}, \ldots, T_n \not\prec T_i^{p+1}, \ldots, T_i^n\}$$

The implicitly disallowed signatures belong to the range of the method but are not covered by it. For invocations with such signatures, the MSA m_i is not a total match. Thus, we also have:

$$\begin{aligned} implicit(m_i) &= range(m_i) - cover(m_i) \\ &= \{(T_1, \ldots, T_n) \in well\text{-}typed(M) \mid m_i \\ &= MSA(m(T_1, \ldots, T_n)) \text{ and } m_i \text{ is not} \\ &\quad \text{a total match for } m(T_1, \ldots, T_n)\} \end{aligned}$$

When all arguments are targetted (i.e., $p = n$), the range of a method m_i is a subset of the signatures covered by m_i. Thus, we have:

Fact 5.1. If a function m is targetted on all arguments, then $implicit(m_i) = \emptyset$ for all of its methods.

Example 5.5. Consider again the method $allocate_2$. We have: $implicit(allocate_2) = \{(Pr, R), (T, R)\}$ and we can see that it is equal to $range(allocate_2) - cover(allocate_2)$
As the explicitly disallowed signatures of a method are in its cover, contrary to its implicitly disallowed signatures, we have the following fact:

Fact 5.2. $\forall m_i,\ implicit(m_i) \cap explicit(m_i) = \emptyset$

Disallowed signatures of a method. The set of *disallowed signatures* of a method m_i, noted $disallowed(m_i)$, is defined as:

$$\forall m_i,\ disallowed(m_i) = explicit(m_i) \cup implicit(m_i)$$

Example 5.6. Applying the above definition to $allocate_2$, we have $disallowed(allocate_2) = \{(T, G), (Pr, R), (T, R)\}$. One can verify in the same way that $disallowed(allocate_1) = \emptyset$.

6 Type checking with exceptions

In this section, we consider the type checking of statements in the presence of exceptions to consistency. To specify type checking we use a generic function called $check$. It has four methods that handles constants, variables, assignments of the form $t \leftarrow e_1$ and invocations of the form $m(e_1, \ldots, e_n)$, where each e_i is an expression. The result of each $check$ method is either *incorrect*, *safe* or *unsafe*. For trivial cases, the result for constants and variables is $safe$.

The last two methods (i.e., for assignments and invocations) proceed in two steps. The first step evaluates the safety of the statement using the types of the expressions e_i known at compile-time; also called the *static types*. If the statement is found to be safe, then its safety is further evaluated in the second step. This step uses the potential types, at run-time, of the expressions e_i composing the statement. These types are called the *dynamic types*.

The distinction between the static and dynamic types is required in the presence of return-exceptions. When covariance of the result types is respected, the type of an invocation known at compile-time is the unique most general type that the invocation may have at run-time. This is not true when a method is allowed to return a type that is not a subtype of the types returned by more general methods. Looking back at Example 3.7, the invocation $doctor(myPatient)$ has $Physician$ for its static return type. However, due to the return-exception $doctor_2$, its possible types at run-time are not only the subtypes of its static type $Physician$, but also the subtypes of $Psychologist$. Thus, its dynamic types are $cover(Physician) \cup cover(Psychologist)$.

This section is organized as follows. First we detail the type checking algorithms for assignments and invocations. They are based on the type checking of *reduced* statements, i.e. statements where the expressions e_i of the input statements are replaced by their static or dynamic types. We then specify the type checking of a reduced statement. Finally, we define the static and dynamic types of expressions.

6.1 Static type checking of assignments

To type check an assignment $v \leftarrow e$, the first step replaces v and e by their static types which are computed by function $static$. The resulting reduced statement is then checked using function $check_R$. If it is incorrect or unsafe, i.e., not safe, then $v \leftarrow e$ is incorrect or unsafe, respectively. Otherwise, its safety must be further probed using the dynamic types of the right-hand side, e. An assignment can be unsafe for two reasons: (1) e is not safe, or (2) e may return, at run-time, a type that is not a subtype of the type of v. The set of most general types that e may evaluate to at run-time is computed using function $dynamics$.

```
check(v ← e) /* check for assignments */
input: an assignment v ← e
output: incorrect, safe or unsafe
Step 1: /* Safety with respect to static types:
           replace v and e by their static type using static */
   reducedAssignment ← ( static(v) ← static(e) ) ;
   result ← check_R( reducedAssignment ) ;
   if result is not safe
      return result ;
Step 2: /* Safety with respect to dynamic types */
   if check(e) is not safe
      return unsafe ;
   /* Replace the right-hand side by each
      of its most general dynamic types using dynamics */
   for each T ∈ dynamics(e) do
      reducedAssignment ← ( static(v) ← T ) ;
      if check_R( reducedAssignment) is not safe
         return unsafe ;
   end do ;
   return safe ;
   end check
```

6.2 Static type checking of invocations

To type check an invocation $m(e_1, \ldots, e_n)$, the first step replaces its arguments which are computed by their static types. The resulting reduced invocation is then checked using function $check_R$. If it is incorrect or unsafe, i.e., not safe, then $m(e_1, \ldots, e_n)$ is incorrect or unsafe, respectively. Otherwise, the invocation is statically correct and its safety must be further evaluated in the second step. At this step, the invocation may be unsafe for two reasons: (1) an unsafe argument e_i exists or (2) for some signature at run-time, the invocation is not safe. Otherwise, the invocation is safe. Function $signatures$ computes the set of most general signatures that may appear as arguments of a method invocation at run-time.

```
check(m(e_1, ..., e_n)) /* check for invocations */
input: an invocation m(e_1, ..., e_n)
output: incorrect, safe or unsafe
Step 1: /* Safety with respect to static types:
           replace arguments by their static type using static */
   reducedInvocation ← ( m(static(e_1), ..., static(e_n))) ;
   result ← check_R(reducedInvocation) ;
   if result is not safe
      return result ;
Step 2: /* Safety with respect to dynamic types */
   for each argument e_i do
      if check(e_i) is not safe
         return unsafe ;
```

```
    end do ;
/*  Using signatures, replace the arguments by each
    of the most general signatures at run-time */
    for each s ∈ signatures(m(e1, ..., en)) do
        reducedInvocation ← m(s) ;
        if check_R (reducedInvocation) is not safe
            return unsafe ;
    end do ;
    return safe ;
    end check
```

6.3 Type checking reduced statements

A reduced assignment is an expression of the form $T_1 \leftarrow \tau_2$, while a reduced invocation is an expression of the form $m(s) = m(\tau_1, \ldots, \tau_n)$. The type checking of reduced assignments is defined as follows.

$$check_R(T_1 \leftarrow \tau_2) = \begin{cases} \text{safe} & \text{if } \tau_2 \preceq T_1 \\ \text{unsafe} & \text{if } (T_1) \in cover(\tau_2) \\ \text{incorrect} & \text{otherwise} \end{cases}$$

$$check_R(m(s)) = \text{incorrect if } \begin{cases} MSA(m(s)) = m_\top \text{ or} \\ MSA(m(s)) \text{ is not a total match for } m(s), \text{ or} \\ s \text{ is explicitly disallowed for } m \end{cases}$$

Note that we allow assignments where the static type of the right-hand side is a supertype of the type of the left-hand side variable. Such unsafe assignments are similar to the reverse assignment of Eiffel (Meyer 1992) or the dynamic downward cast of C++ (Lajoie 1993). The safety of a reduced invocation is defined as follows:

$$check_R(m(s)) = \begin{cases} \text{safe iff } \forall s' \in cover(s) \;\; check_R(m(s')) \\ \qquad \neq \text{ incorrect} \\ \text{unsafe otherwise} \end{cases}$$

We now give the algorithm to type-check reduced invocations:

```
check_R(m(s))
input: a reduced invocation m(s)
output: incorrect, safe or unsafe
    msa ← MSA(m(s)) ;
Step 1: /* Check the correctness */
    if msa = m_⊤ or msa is not a total match or s ∈ explicit(msa)
        return incorrect ;
Step 2: /* Check the safety */
    for each s' ∈ cover(s) do
        msa' ← MSA(m(s')) ;
        if msa' = m_⊤ or msa' is not a total match or s'∈ explicit(msa')
            return unsafe ;
    end do ;
    return safe ;
    end check
```

6.4 Static and dynamic types of an expression

The static type of an expression can now be defined as shown on Fig. 20.

Example 6.1. Consider again the types and methods of Fig. 8 of Sect. 3. Let $refund(\underline{Hospital}, \underline{Dollar})$ be the method used in Example 3.7 to refund the expenses of patients to hospitals. The first step in the type-checking of invocation $refund(hospital(\ doctor(p)), amount)$, where p is a variable of type $Patient$ and $amount$ a variable of type $Dollar$, consists of computing the static types of the arguments $hospital(doctor(p))$ and $amount$ as follows:

$$static(hospital(doctor(p))) =$$
$$static(hospital(static(doctor(p)))) =$$
$$static(hospital(static(doctor(static(p))))) =$$
$$static(hospital(static(doctor(Patient)))) =$$
$$static(hospital(Physician)) = Hospital$$

and $$static(amount) = Dollar$$

As $check(refund(Hospital, Dollar)) \neq$ incorrect, invocation $refund(hospital(doctor(p)), amount)$ is correct.

We now formally define the dynamic types of an expression as shown in Fig. 21. The set of dynamic types of a reduced invocation contains only the highest types that can be returned by the invocation at run-time. By highest, we mean types that are not subtypes of any other type in the set (we use the operator $max_\preceq$ to obtain the highest types in a set of types).

The definition of the dynamic types of a reduced invocation $m(s)$ relies on the notion of *run-time correct* methods. They represent the methods that can be selected at run-time for *correct* invocations covered by $m(s)$.

Run-time correct methods. Let $m(s)$ be a reduced invocation.

$$RTC(m(s)) = \{MSA(m(s')) \mid s' \in cover(s) \text{ and } check(m(s')) \neq \text{ incorrect}\}$$

The definition of the dynamic types of an invocation $m(e_1, \ldots, e_n)$ relies on the set of signatures that may appear at run-time as arguments of the invocation. As usual, this set contains only the highest signatures; all the signatures in their cover being implicitly included. This set is denoted *signatures*$(m(e_1, \ldots, e_n))$ and consists of the cross product of the dynamic types of the invocation's arguments:

Signatures of an invocation. The set of highest signatures that may appear at run-time for an invocation is:

$$signatures(m(e_1, \ldots, e_n)) = \prod_{i=1}^{n} dynamics(e_i)$$

Example 6.2. The second step in the type checking of the invocation $refund(hospital(doctor(p)), amount)$ starts by type checking $hospital(doctor(p))$ and $amount$. First, $hospital(static(doctor(p))) = hospital(Physician)$ is neither incorrect or unsafe. Thus the safety of $hospital(doctor(p))$ must be checked. To this end, the algorithm determines the signatures of $hospital(doctor(p))$.

$$signatures(hospital(doctor(p))) = \{(T) \mid T \in dynamics(doctor(p))\} = \{(Physician), (Psychologist)\}$$

One of the signatures of $hospital(doctor(p))$, namely $Psychologist$, makes the invocation incorrect as there is no MSA method. Thus $hospital(doctor(p))$ is unsafe. So

Constant c	$static(c) = \overline{T}$
Variable v	$static(v) = T$
Reduced Invocation $m(s)$	$static(m(s)) = \begin{cases} T_\top \text{ if } check(m(s)) = \text{ incorrect} \\ \text{return type of } m_k = MSA(m(s)) \text{ otherwise} \end{cases}$
Invocation $m(e_1, \ldots, e_n)$	$static(m(e_1, \ldots, e_n)) = static(m(static(e_1), \ldots, static(e_n)))$

Fig. 20. Static type of expressions

Constant c	$dynamics(c) = \{\overline{T}\}$
Variable v	$dynamics(v) = \{T\}$
Reduced Invocation $m(s)$	$dynamics(m(s)) = max_{\preceq}\{R_i \mid m_i \in RTC(m(s))\}$
Invocation $m(e_1, \ldots, e_n)$	$dynamics(m(e_1, \ldots, e_n)) = max_{\preceq}(\bigcup_{s \in signatures(m(e_1,\ldots,e_n))} dynamics(m(s)))$

Fig. 21. Dynamic types of an expression

finally, as one of its arguments is unsafe, $refund(hospital(doctor(p)), amount)$ is unsafe.

7 Optimizing the type checking of reduced invocations

In this section, we propose an optimization of the type-checking of reduced invocations. The algorithm for $check_R$ presented in Sect. 6.3 is expensive because it requires to compute the MSA method for every signature in the cover of the reduced invocation. Optimizing $check_R$ is particularly important, as it is called several times by $check$ to type check a general invocation. The idea of the optimization is the following. Given a reduced invocation, if no signature in its cover, i.e., the run-time signatures, is a disallowed signature of some method, then the invocation is safe. To evaluate this condition, one computes the set of disallowed signatures of the methods that are more specific than the MSA method of the reduced invocation. This set is called the *potential disallowed signatures* of the MSA method. If a method has no potential disallowed signatures, then all invocations, for which it is the MSA method are safe. Such a property of a method is called *static safety* and constitutes a cheap sufficient condition for the safety of a reduced invocation.

In this section, we first give the optimized algorithm, and then present the two safety conditions that it uses.

7.1 Optimized algorithm for the static type checking of reduced invocations

The first step of the optimized algorithm checks the correctness of a reduced invocation, following the same criteria as in Sect. 6.3 on the MSA method of the invocation. Steps 2 and 3 check the safety. Step 2 checks whether the MSA method of the invocation is static safe. If it is not, Step 3 verifies that no run-time signature, i.e., no signature covered by the invocation, is a potential disallowed signature of the MSA method.

$check_R(m(s))$
input: a reduced invocation $m(s)$
output: $incorrect$, $safe$ or $unsafe$
  $msa \leftarrow MSA(m(s))$;
Step 1:
  if $msa = m_\top$ or msa is not a total match or $s \in explicit(msa)$
    return **incorrect** ;
Step 2:
  if msa is static safe
    return **safe** ;
Step 3:
  if no signature in $cover(m(s))$
   is a potential disallowed signature of msa
    return **safe** ;
  return **unsafe** ;
  end check

7.2 Safety and potential disallowed signatures

The third step of the algorithm relies on Proposition 1 below, which gives a necessary and sufficient condition for the safety of a correct reduced invocation.

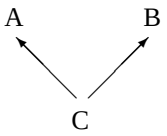
Potential disallowed signatures of a method. The set of potential disallowed signatures of a method m_i, noted $potential\_disallowed(m_i)$, is defined as:

$$potential\_disallowed(m_i) = \bigcup_{\substack{m_j \leq_s m_i \\ s \in cover(m_i)}} disallowed(m_j)$$

Proposition 1. *A correct reduced invocation $m(s)$ is safe iff* $potential\_disallowed(MSA(m(s))) \cap cover(s) = \emptyset$

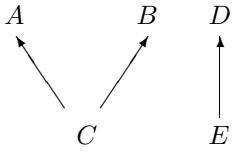
Proof. see Appendix 11.

Example 7.1. Consider the hierarchy and methods of Fig. 22. The potential disallowed signature of m_1 is (C, A), and there is no potential disallowed signature for m_2. Invocation $m(A, A)$ is declared unsafe, because the signature (C, A) is covered by it.



$m_1(\underline{A},\underline{A})$
$m_2(\underline{B},\underline{B})$ $\quad m_2 < m_1$
$explicit(m_1)$=(C, A)

Fig. 22. Safety conditions



$m_1(\underline{A},D) \rightarrow E \quad m_2 < m_1$
$m_2(\underline{B},E) \rightarrow D \quad m_3 < m_2$
$m_3(\underline{C},D) \rightarrow E \quad m_3 < m_1$

Fig. 23. Consistent schema

7.3 Static safety of a method

The second step of the algorithm relies on Proposition 2 below, which gives a sufficient condition for the safety of a correct reduced invocation. It uses the notion of *static safety* of a method. This property is invocation-independent and may be computed once for each method, at compile-time.

Static safety of methods. A method m_i is *static safe* iff

$potential\_disallowed(m_i) \cap cover(m_i) = \emptyset$

As $\forall s,\ cover(s) \subseteq cover(MSA(m(s)))$, we have:

Proposition 2. *$MSA(m(s))$ is static safe $\Rightarrow$ $m(s)$ is safe.*

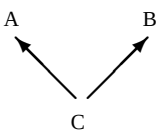
Proof. As $cover(s) \subset cover(MSA(m(s)))$, we have $potential\_disallowed(MSA(m(s))) \cap cover(MSA(m(s))) = \emptyset \ \Rightarrow potential\_disallowed(MSA(m(s))) \cap cover(s) = \emptyset$, which implies the safety from Proposition 1.

Example 7.2. In Fig. 22, the method m_2 has no potential disallowed signature, thus it is static safe. The invocation $m(A, C)$ is safe even though its MSA method, m_1, is not static safe, and $m(B, C)$ is safe.

8 Safety and consistency

In this section, we establish the relation between safety and consistency, introducing the notion of the *trespassing* method. A method m_i trespasses on method m_j, if m_i may be selected at run-time for invocations whose MSA method at compile-time is m_j. We show that argument-exceptions and return-exceptions cause safety problems only if they are coupled with trespassing. If m_i is an argument- or return-exception to m_j, but may not be selected for invocations whose MSA method at compile-time is m_j, then no run-time type error may occur.

Example 8.1 Consider the schema of Fig. 23 with method $foo_1(E) \rightarrow E$. The MSA method of invocation $m(a, d)$ is m_1. As method m_2 is both a return-exception and an argument-exception to m_1, a run-time type error could occur if m_2 was selected at run-time. For example, $m(a, d)$ could return a D and there would not be any foo method applicable to the invocation $foo(m(a, d))$. However, all invocations for which both m_1 and m_2 are applicable, namely $m(C, D)$ and $m(C, E)$, have m_3 as their MSA method . And m_3 is consistent with respect to m_1 and m_2. Thus, the inconsistency of m_2 with respect to m_1 cannot lead to any safety problem, because m_2 never trespasses on m_1.



$m_1(\underline{A},A)$
$m_2(\underline{B},B)$
$m_2 < m_1$

Fig. 24. Trespassing

We first give the definition of a trespassing method, then a proposition that states the relationship between trespassing, consistency and safety.

Trespassing. A method m_i *trespasses* on method m_j iff $range(m_i) \cap cover(m_j) \neq \emptyset$

Example 8.2. In the example of Fig. 24, (C, A) and (C, C) are both in $range(m_2)$ and in $cover(m_1)$. Thus, m_2 trespasses on method m_1.

Proposition 3. *If m_i is a return- and/or argument-exception to m_j, then a run-time type error may occur only if m_i trespasses on m_j.*

Proof. see Appendix 11.

Example 8.3. Looking back at Fig. 23, we see that there is no method m_i that is both an argument-exception to some method m_j and trespasses on it: m_2 is an argument-exception to m_1, but does not trespass on it, while m_3 trespasses on both m_1 and m_2, but is not an argument-exception to them. Thus, $implicit(m_i) = \emptyset$, for all m_i. Note that removing m_3 makes m_2 trespassing on m_1, so that $implicit(m_2) = \{(C, D)\}$.

Moreover, there is no method m_i that is both a return-exception to some method m_j and trespasses on it: m_2 is a return-exception to m_1, but does not trespass on it, while m_3 trespasses on both m_1 and m_2, but is not a return-exception to them. Thus, the type returned at run-time by any well-typed invocation $m(s)$ is guaranteed to be a subtype of the static type of $m(s)$.

It must be noted that although the covariance and contravariance rules are too pessimistic, they are adopted in most systems because they are simpler to check and they offer a better support for schema evolution. Indeed, adding or removing a method that abides by the covariance and contravariance rules with respect to all other methods, has no consequences on safety. As we showed in the above examples, this is not the case when trespassing is taken into account.

9 Final steps

The last two steps of the type checking process are the generation of check statements and the type checking of the exception-handling code provided by the user. These two issues are out of the scope of this paper. In this section, we just give an idea of the problems and sketch the solution.

9.1 Generation of check statements

As invocations may appear as arguments of other invocations, a single statement may contain several unsafe subexpressions. This naturally leads to nest the check statements.

Example 9.1. We re-use the methods and types of Fig. 8 and assume the following methods exists: $bill(Patient) \rightarrow Dollar$, to get the expenses of a patient and $refund(Hospital, Dollar)$ to refund hospitals for the expenses of their patients. Moreover, we assume that alcoholics are not billed for their treatment, i.e., $Alcoholic$ is an explicitly disallowed signature of $bill(Patient)$. Consider statement $refund(hospital(doctor(p)), bill(p))$. A naive approach to check generation examines each subexpressions of a statement in a left-to-right, depth-first order and generates a check statement whenever an unsafe subexpression is encountered. Thus, after type checking p,$doctor(p)$, $hospital(doctor(p))$, p, $bill(p)$, the following nested check statements are generated:

```
CHECK hospital IS CORRECT ON (doctor(p))
  CHECK bill IS CORRECT ON p
  ELSE
  END
ELSE
END
```

However, the check on $bill(p)$ is redundant. Indeed, if $hospital(doctor(p))$ is correct, then $doctor(p)$ is a $Physician$ and p is not an $Alcoholic$. Thus, $bill(p)$ is safe.

To correctly generate check statements, the idea is to provide the type checker with the ability to infer the run-time types of subexpressions like $doctor(p)$ and p, based on the previously generated checks. The inferred types are then used to bind the remaining subexpressions, using what can be called *type closures*. These bindings are then used by the type checker.

9.2 Type checking the exception-handling code

Type checking the exception-handling code provided by the user differs from the type checking we defined in the two previous sections. To give an idea of the problem, consider the following example:

Example 9.2. Going back to the doctor and patient hierarchy of Fig. 8, assume there exists a method $practice(Psychologist) \rightarrow Office$ to access the practice of psychologists and two methods $address(Hospital) \rightarrow Address$ and $address(Office) \rightarrow Address$ to get the addresses. Consider the exception-handling code of the following check statement:

```
addr : Address;
CHECK hospital IS CORRECT ON (doctor(p))
   addr ← address(hospital(doctor(p)));
ELSE
   addr ← address(practice(doctor(p)));
END
```

In the ELSE part, one can infer that $doctor(p)$ is of type $Psychologist$. This allows to write a modified version of the original statement using $practice$ instead of $hospital$. However, note that the ELSE statement is not correct according to standard static type checking, as $static(doctor(p))$ = $Physician$ and $practice$ is not applicable to $Physician$.

As for the generation of check statements, the solution is to provide the type checker with the ability to infer the run-time types of subexpressions like $doctor(p)$ based on the previously generated checks.

10 Related work

The problems due to maintaining consistency rules have been recognized by many researchers, each focusing on a particular rule, but to our knowledge, considering these problems in a single framework has never been proposed.

Cook (1989), Mc Kenzie (1992), and Danforth and Simon (1992) forbid argument-exceptions. Hence, subtyping between generic collections (list of $Person$ and list of $Student$) and attribute type redefinition are also disallowed.

Esse (Coen-Porsini et al. 1991; Cattaneo et al. 1993) and Eiffel (Meyer 1992) use data flow analysis to detect unsafe invocations due to argument-exceptions: the set of types to which a variable may refer (called *type set* by Coen-Porsini et al. (1991) and Cattaneo et al. (1993) and the *dynamic class set* in Meyer (1992)) are maintained during type checking and evaluated after every statement. Using this "type flow" technique, a slightly larger class of programs are statically determined to be safe, as exact types may be used to replace constant objects or variables that have just been assigned a newly-created object. Although this approach provides more accurate type checking, two problems remain. First, statements that cannot be proved to be safe are rejected (pessimistic option). Second, this approach is less applicable to a database context where applications use collections. Indeed, a variable iterating over a collection of T may refer to objects of any subtype of T with no way of knowing the exact subset of types present in the collection. Our approach can be used as a complement to "type flow" techniques, taking over when they have failed to prove the safety of a statement.

Using a special construct called *reverse assignment*, Eiffel (Meyer 1992) allows a certain kind of illegal substitution: the assignment of an expression with static type T_1 to a variable of type T_2, although T_1 is a supertype of T_2. The assignment is checked at run-time to ensure that the dynamic type of the expression is actually T_2 or a subtype of T_2. Otherwise, a NULL reference is assigned to the variable. It is the responsibility of the programmer to check that the variable is not NULL after the reverse assignment. A similar construct, the *dynamic cast* (Lajoie 1993), is being incorporated into C++ to check, at run-time, the correctness of a *down-ward cast* (assertion by the programmer that an object of static type T_1 is actually of type T_2 with T_1 supertype of T_2).

Bounded type quantification, first introduced by Cardelli and Wegner (1985), appears in several proposals (Connor and Morrison (1992); Canning et al. 1989; Kemper and Moerkotte 1994) to extend the flexibility of statically typed object-languages. As explained by Kemper and Moerkotte (1994), it enables "polymorphic operations [. . .] to deal with objects of different types that do not necessarily lie on the

same branch of the super/subtype relationship". Connor and Morrison (1992) uses bounded type quantification, restricting the application of subtyping to enforce the composition integrity constraint on constructed types. Bounded universal quantification allows substitutability only when passing parameters to a function. All other assignments must involve objects of the same type. Bounded existential quantification extends substitutability to assignments in the called function. In all cases, bounded quantification requires the exact types of actual parameters to be known statically. It is this knowledge that allows static type checking of covariant code. In particular, this prevents passing bounded parameters to another function. Finally, F-bounded quantification (Canning et al. 1989) allows support of recursively defined types, like $Person$ and $Student$ in Fig. 1.

In the works on *method schemas* (Abiteboul et al. 1990; Walter 1991), no consistency rules are imposed on the schema and the return type of user-defined methods is not specified. Consistency is defined as type safety, i.e., absence of run-time type errors. Proving type safety involves simulating the execution of methods from a typing point of view. This is shown to be impossible in the general case, i.e., with multi-targeted methods and recursion. Covariant updates are shown to maintain consistency.

Madsen et al. (1990) recognize the conflict that arises from the use of the type system both "as a means for representing concepts in the application domain and for detecting [...] type errors". They show that the subtyping of "virtual classes" (i.e., classes with a type parameter) introduces type holes, similar to component type redefinition. They conclude that a combination of compile-time and run-time type checks, as implemented in Beta, gives a good balance of flexibility and type safety. All operations on virtual classes involve run-time type checking. Furthermore, an error occurs if a statement in a Beta program is not correct at run-time.

Our approach is very similar to Borgida's (1988) in that it aims at detecting unsafety at compile-time, using dynamic type checking when necessary and allowing the user to write exception handling code. Borgida (1988) addresses the problem of inapplicable attributes and return-exceptions due to attribute domain redefinition. The notion of *excuses* serves to distinguish between desired exceptions and errors. A type system that supports these excuses is formally defined by Borgida (1989), along with an efficient algorithm to statically detect unsafe statements. Check clauses are provided by the user. The user formulates the correction condition in an extensional way, testing the run-time type of expressions. The type system verifies that the correction condition implies the safety of the checked statement and of the exception-handling code. We extend this work in two directions. First, we address the problem of exceptions on single- and multi-targeted methods. Second, we provide an intensional formulation of the correction condition, allowing this condition to remain invariant when the type hierarchy is modified and/or new exceptions are introduced.

11 Conclusion

In this paper, we proposed to facilitate schema evolution in object-oriented databases by supporting exceptions to behavioral schema consistency, while at the same time guaranteeing type safety. After presenting the three consistency rules of covariance, contravariance and substitutability, we defined a typology of exceptions. We gave examples of schema updates that naturally yield exceptions to the consistency rules, and we showed that existing solutions that seek preserving schema consistency lead to expensive modifications of the type hierarchy and method codes. We then proposed a new type checking process whereby exceptions to consistency can be safely tolerated. To guarantee type safety, every statement is first analyzed to determine whether it is correct, and then further analyzed to determine whether it is safe. Then, every unsafe statement is surrounded by a check clause. This clause is merely an if-then-else statement where the if-part performs a run-time type checking, the then-part contains the original statement, and the else-part contains some exception-handling code (user-defined or system-generated).

Unlike traditional solutions offered by object-oriented design, our approach enables the handling of schema updates that do not preserve schema consistency without creating artificial types and methods, or modifying the type hierarchy. Schema updates can only yield the additions of check clauses in the code of existing methods. Another advantage of our solution is that conditions in the check are specified intensionally, thereby avoiding their reformulation when the type hierarchy is modified, or when exceptions are introduced or removed. We believe our approach provides a useful complement to existing sophisticated techniques for static type checking. Indeed, our proposed system relieves these techniques when they fail to prove the safety of a statement. Finally, we are not aware of any other work in the field of object-oriented systems and languages that consider exceptions to schema consistency in the general framework of mono- and multi-targeted functions.

All the steps of the proposed type checking process have now been specified (see Amiel 1994). Future work involves providing the user with means to express explicitly disallowed signatures, and developing efficient algorithms to implement our type checking. Finally, an environment to help programming with exceptions is being designed. Such an environment addresses important issues, such as providing the user with explanations about why some statements are unsafe and assistance in writing exception-handling code.

Acknowledgement. We would like to thank François Bancilhon and Guy Ferran for their interest in this work. Special thanks go to Catriel Beeri, Françoise Fabret, Claude Delobel and Patrick Valduriez for their insightful comments on an earlier version of this paper. We also would like to thank the three referees for their extremely careful reviews and helpful comments.

Appendices

A. Proof of Proposition 1

We first introduce the following lemma:

Lemma 1. $\forall s, s',\ s' \in cover(s) \Rightarrow MSA(m(s')) \leq_{s'} MSA(m(s))$

Proof. As $s' \preceq s$, $MSA(m(s))$ is applicable to s'. As $MSA(m(s'))$ is the most specific method applicable to s'

with respect to the ordering of s', $MSA(m(s'))$ is more specific or equal to $MSA(m(s))$.

We prove that a necessary and sufficient condition of safety on $m(s)$ is that $potential\_disallowed(MSA(m(s)))$ $\cap cover(s) = \emptyset$. This amounts to the following equivalence:

$m(s)$ is correct and

$$\Big(\bigcup_{\substack{m_i \leq_{s''} MSA(m(s)) \\ s'' \in cover(MSA(m(s)))}} disallowed(m_i)\Big) \cap cover(s) = \emptyset \quad (1)$$

equivalent to

$\forall s' \in cover(s),\ MSA(m(s')) \neq m_\top$ and
$MSA(m(s'))$ is a total match for $m(s')$ and
$s' \notin explicit(MSA(m(s')))$ (2)

Using the definition of the correctness of a reduced invocation, (1) can be written:

$MSA(m(s)) \neq m_\top$ and
$MSA(m(s))$ is a total match for $m(s)$ and
$\forall s' \in cover(s),\ \forall s'' \in cover(MSA(m(s)))$,
$\forall m_i \leq_{s''} MSA(m(s)), s' \notin disallowed(m_i)$ (3)

To rewrite (2), we use the following equivalence, that comes from the definition of $implicit(m_i)$:

$MSA(m(s'))$ is a total match
for $s' \Leftrightarrow s' \notin implicit(MSA(m(s')))$

Thus (2) can be written:

$\forall s' \in cover(s),\ MSA(m(s')) \neq m_\top$ and
$s' \notin disallowed(MSA(m(s')))$ (4)

Let us prove now that (3) $\Rightarrow$ (4). We assume that (3) is true for m and s. $MSA(m(s)) \neq m_\top$ implies $\forall s' \in cover(s),\ MSA(m(s')) \neq m_\top$. Thus the first conjunct of (4) is established.

We also prove that $\forall s' \in cover(s),\ s' \notin disallowed(MSA(m(s')))$, applying the second conjunct of (3). From Lemma 1, we have $MSA(m(s')) \leq_{s'} MSA(m(s))$, and $s' \in cover(s)$ implies $s' \in cover(MSA(m(s)))$. Thus $s' \notin disallowed(MSA(m(s')))$. This concludes the first part of our proof.

Let us prove now that (4) $\Rightarrow$ (3). We assume that (4) is true for m and s. As $s \in cover(s)$, we have that $MSA(m(s)) \neq m_\top$. As $s \notin implicit(MSA(m(s)))$, $MSA(m(s))$ is a total match for $m(s)$.

Now let $s' \in cover(s)$, $s'' \in cover(MSA(m(s)))$, and $m_i \leq_{s''} MSA(m(s))$. If $m_i = MSA(m(s'))$, from (4) we have $s' \notin disallowed(m_i)$. If $m_i \neq MSA(m(s'))$, from the definition of $implicit(m_i)$ and $explicit(m_i)$, $disallowed(m_i) \in range(m_i)$, thus $s' \notin disallowed(m_i)$. This concludes the proof of proposition 1.

B. Proof of Proposition 3

We prove that a run-time type error due to a method m_i being a return- or an argument-exception to a method m_j, may occur only if m_i trespasses on m_j. We first consider the case of return-exceptions, then of argument-exceptions.

A run-time type error may occur due to a method m_i being a return-exception to a method m_j, iff for some static signature $s \in well\text{-}typed(m)$, $m_i \in RTC(m(s))$ and $m_j = MSA(m(s))$. We have to prove that in this case, m_i trespasses on m_j.

As $m_i \in RTC(m(s))$, there exists $s' \in cover(s)$ such that $m_i = MSA(m(s'))$. As $s \in cover(m_j)$, we have $s' \in cover(m_j)$, and $s' \in range(m_i)$, thus $range(m_i) \cap cover(m_j) \neq \emptyset$. This concludes the first part of our proof.

Let us now consider the case of argument-exceptions. The general static safety condition is:

$potential\_exceptions(m_i)) \cap cover(m_i) = \emptyset$

Using the decomposition of exceptions into implicit and explicit exceptions, we can rewrite the condition as:

$$\Big(\Big(\bigcup_{m_j \leq m_i} implicit(m_j)\Big) \cup \Big(\bigcup_{m_j \leq m_i} explicit(m_j)\Big)\Big) \cap cover(m_i) = \emptyset \ \Leftrightarrow$$
$$\Big(\bigcup_{m_j \leq m_i} implicit(m_j)\Big) \cap cover(m_i) = \emptyset, \text{ and}$$
$$\Big(\bigcup_{m_j \leq m_i} explicit(m_j)\Big) \cap cover(m_i) = \emptyset$$

Static safety with respect to implicit exception corresponds to the first part of the conjunction. As the implicit exceptions of methods are due to argument exceptions, let us see what conditions must hold on m_i and the $m_j < m_i$.

As $implicit(m_i) \cap cover(m_i) = (range(m_i) - cover(m_i)) \cap cover(m_i) = \emptyset$, the first conjunctive term can be written as:

$$\bigcup_{m_j < m_i} implicit(m_j) \cap cover(m_i) = \emptyset \quad (5)$$

Let us prove that if (5) is false, then $\exists m_j < m_i$ such that m_j is an argument-exception to m_i and trespasses on m_i

For this, we show that if either (a) m_j is not an argument-exception to m_i, or (b) m_j does not trespass on m_i, then (c) $implicit(m_j) \cap cover(m_i) = \emptyset$.

Let us first prove (A) $\Rightarrow$ (C). (A) means that:

$$T_j^{p+1}, \ldots, T_j^n \succeq T_i^{p+1}, \ldots, T_i^n \quad (6)$$

For all $T^1, \ldots, T^n$ in $implicit(m_j)$, we have:

$$T^{p+1}, \ldots, T^n \not\preceq T_j^{p+1}, \ldots, T_j^n \quad (7)$$

(6) and (7) imply:

$$T^{p+1}, \ldots, T^n \not\preceq T_i^{p+1}, \ldots, T_i^n \quad (8)$$

And thus:

$m_j$not argument-exception to m_i
$\Rightarrow implicit(m_j) \cap cover(m_i) = \emptyset$ (9)

Let us now prove (b) $\Rightarrow$ (c). We have that $implicit(m_j) = range(m_j) - cover(m_j)$ and (b) means that $range(m_j) \cap cover(m_i) = \emptyset$. Thus, $implicit(m_j) \cap cover(m_i) = \emptyset$. This concludes the proof of Proposition 3.

References

1. Abadi M, Cardelli L, Pierce B, Plotkin G (1989) Dynamic typing in a statistically-typed language. In: Proc POPL, Austin, Tex, January
2. Agrawal R, DeMichiel LG, Lindsay BG (1991) Static type checking of multi-methods. In: Proceedings of the Sixth International Conference on Object-Oriented Programming: Systems, Languages and Applications, Phoenix, Ariz, October
3. Abiteboul S, Kanellakis P, Waller E (1990) Method schemas. In: Proceedings of the Fourth ACM Symposium on the Principles of Database Systems, Nashville, 10 April
4. Amiel E (1994) Schémas orientés-objet: Exceptions à la cohérence comportementale et envoi de multi-méthodes. PhD thesis, University of Paris VI
5. Apple Computer (1994) Dylan interim reference manual, June. Available by ftp from ftp.cambridge.apple.com in /pub/dylan/dylan-manual
6. Banerjee J, Kim W, Kim HJ, Korth HF (1987) Semantics and implementation of schema evolution in object-oriented databases. In: Proceedings of the ACM SIGMOD International Conference on Management Of Data, San Francisco, Calif, June
7. Bobrow DG, Kahn K, Kiczales G, Masinter L, Stefik M, Zdybel F (1986) CommonLoops: merging Lisp and object-oriented programming. In: Proceedings of the First International Conference on Object-Oriented Programming: Systems, Languages and Applications, Portland, Ore, September
8. Bobrow DG, DeMichiel LG, Gabriel RP, Keene S, Kiczales G, Moon DA (1988) Common Lisp Object System specification. SIGPLAN Not, 23, September
9. Borgida A (1988) Modeling class hierarchies with contradictions. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Ill, June
10. Borgida A (1989) Type systems for querying class hierarchies with non-strict inheritance. In: Proceedings of the Eigth ACM Symposium on Principles of Database Systems, Philadelphia, Pa, March
11. Bruce KB (1993) Safe type-checking in a statically-typed object-oriented programming language. In: Proc POPL, January
12. Canning P, Cook W, Hill W, Olthoff W (1989) F-bounded polymorphism for object-oriented programming: In Proceedings of the International Conference on Functional Programming and Computer Architecture, London, UK, September
13. Cardelli L (1984) A semantics of multiple inheritance. In: Proceedings of the Symposium on Semantics of Data Types. (Lecture notes on computer science, vol 173) Springer, Berlin Heidelberg New York
14. Cardelli L, Wegner P (1985) On understanding types, data abstraction, and polymorphism. ACM Comput Surv, 17:471–522
15. Cattaneo F, Coen-Porisini A, Lavazza L, Zicari R (1993) Overview and progress report of the ESSE project: supporting object-oriented database schema analysis and evolution. In: Proceedings of the Eleventh International Conference on Technology of Object-Oriented Languages and Systems, Santa Barbara, Calif, August
16. Cattell RGG (ed) (1994) The object database standard: ODMG-93. Morgan Kaufmann, San Mateo, Calif
17. Chambers C (1992) Object-oriented multi-methods in Cecil. In: Proceedings of the Sixth European Conference on Object-Oriented Programming, Utrecht, Netherlands, June
18. Coen-Porisini A, Lavazza L, Zicari R (1991) Updating the schema of an object-oriented database. IEEE Data Eng Bull, 14:33–37
19. Connor RCH, Morrison R (1992) Subtyping without tears. In: Proceedings of the Australian Computer Science Conference, Sydney, February
20. Connor RCH, McNally D, Morrison R (1991) Subtyping and assignment in database programming languages. In: Proceedings of the Second International Workshop on Database Programming Languages, Gleneden Beach, Ore, August
21. Cook W (1989) A proposal to make Eiffel type-safe. In: Proceedings of the Third European Conference on Object-Oriented Programming, Nottingham, UK, July
22. Danforth S (1990) Multi-targetted virtual functions for OODG. In: INRIA, Proceedings of the Sixth Journées Bases de Donnees Avancées, Montpellier, France, September
23. Danforth S, Simon E (1992) A data and operation model for advanced database systems. In: Papazoglou MP, Zeleznikow J (eds) The next generation of information systems: from data to knowledge. (Lecture notes in computer science, vol 611)
24. DeMichiel LG, Chamberlin DD, Lindsay BG, Agrawal R, Arya M (1993) Polyglot: extensions to relational databases for sharable types and functions in a multi-language environment. In Proceedings of the International Conference on Data Engineering, Vienna, Austria, April
25. Formica A, Missikoff M (1994) Correctness of isa hierarchies in object-oriented database schemas. In: Proc EDBT, Cambridge, March
26. Kemper A, Moerkotte G (1994) Object- oriented database management: applications in engineering and computer science. Prentice-Hall, Englewood Cliffs, NJ
27. Lajoie J (1993) The new language extensions. C++ Rep, 5:47–52
28. Madsen OL, Magnusson B, Moller-Pedersen B (1990) Strong typing of object- oriented languages revisited. In: Proc ECOOP- OOPSLA
29. McKenzie R (1992) An algebraic model of class, inheritance, and message passing. PhD thesis, Computer Science Dept., University of Texas at Austin
30. Melton J (ed) (1994) ISO Working Draft. SQL persistent stored modules (SQL/PSM). ANSI X3H2-94-331, August
31. Meyer B (1992) EIFFEL: the language. Prentice-Hall, Englewood Cliffs, NJ
32. Mugridge WB, Hamer J, Hosking JG (1991) Multi-methods in a statistically-typed programming language. In: Proceedings of the Fifth European Conference on Object-Oriented Programming, Geneva, Switzerland, July
33. O2 Technology (1992) The O2 user's manual
34. Waller E (1991) Schema updates and consistency. In Delobel C, Kifer M, Masunaga Y (eds) Proceedings of the Second International Conference on Deductive and Object-Oriented Databases. (Lecture notes in computer science, vol 566) Springer, Berlin Heidelberg New York
35. Zdonik S, Maier D (1989) Fundamentals of object-oriented databases. In: Zdonik S, Maier D (eds) Readings in object-oriented databases. Morgan-Kaufmann, San Mateo, Calif, pp 1–32