1998 International Computer Symposium
Workshop on Software Engineering and Database Systems
December 17-19, 1998, N.C.K.U., Tainan, Taiwan, R.O.C.

Type Checking Issues for Dynamically Changing Types in Object-Oriented Database Systems

Raymond K. Wong
Basser Department of Computer Science
University of Sydney
NSW 2006, Australia
wong@cs.usyd.edu.au

H. Lewis Chau
Department of Computer Science
Hong Kong University of Science & Technology
Clear Water Bay, Hong Kong
lewis@cs.ust.hk

Abstract

The association between an instance and a class is exclusive and permanent in many class-based, object-oriented (OO) database systems. Therefore, these systems have serious difficulties for applications in which objects take on different and multiple roles over time. Recently, some researchers have tried to relax this restriction by allowing an object to have multiple most specific types and be able to change to different types during its lifetime. However, although many of these researchers have realized the importance and difficulties of solving the *type* problems caused by their proposed extensions, formal semantics and corresponding type issues have not been addressed. In particular, a type checking mechanism, which is to ensure the correctness of program writing from the type aspect, has never been proposed for these kinds of OO extensions. In this paper, we develop an expressive yet semantically sound type calculus for objects with multiple types (well them roles). While the rich modeling constructs are introduced, the calculus can still be expressed in a neat way, by using the polymorphism of overloaded functions. Both static and dynamic type checking frameworks are included and discussed individually. Furthermore, we show the important properties of our calculus, which include Subject Reduction, Strong Normalization and Confluence. The calculus described in this paper provides a foundation for dynamic type changing and for objects with multiple most-specific types. We show that the calculus is general enough to be applied to various dynamic type and role models with little modification.

1. Introduction

Most object-oriented data models are based on the notion of class. In these models, real-world entities are represented as instances of the most specific class and the association between an instance and a class is exclusive and permanent. However, objects often belong to several most specific classes in reality, and change their classes during their lifetime. For example, a person can be a graduate student, a teaching assistant and research assistant, a club-member and club-chairman at the same time or from time to time. Thus, the object representing this person does not have a unique and fixed most specific class; rather it has a changing set of most specific classes. Although this situation can be easily represented in a model with multiple inheritance by defining a subclass of all the involved classes, this solution may lead to a combinatorial explosion of artificial subclasses and the number of possible involved classes can be enormous. Moreover, multiple inheritance only provides a single behavioral context for an object [16].

Recently, some researchers have proposed different approaches (but similar ideas) to relax these restrictions [3, 5, 16, 18]. In particular, many of them have used the notion of roles. A role extends an existing object with additional states and behaviors. An object may have many roles that come and go over time. Rather than being an instance of some unique subclass defined through multiple inheritance, an object simply is an instance of many types by virtue of having many roles. Every object reference is to a particular role. Its behavior depends on which role is being referenced.

The concept of a role was already defined in 1977 by Bachman and Daya [4] in the context of the network data modeling approach. Various role models and implementations in the context of databases were proposed subsequently. These include Vision [17], ORM [15], and aspects [16], etc., and the most recent systems include Fibonacci [3], Gottlob *et al.*'s role extension of Smalltalk [12], and DOOR [19]. Fibonacci is a new, strongly-typed database language. Its objects simply consist of an identity and an acyclic graph of roles. Each role can be dynamically added or dropped. Objects are defined in classes and roles are defined separately and form a different hierarchy. Instead of implementing a new language, Gottlob *et al.* [12] demonstrated the extension of Smalltalk for incorporating roles. In contrast to Fibonacci, they included multiple instantiation of roles, and the integration of class and role hierarchies. To some extent, both Fibonacci and Gottlob *et al.*'s work is similar to ORM in the sense that roles are also rooted in (though not encapsulated into) a class, and these roles can be inherited from the class to its subclasses. Different from roles in ORM, aspects and views, however, the roles attached to a class in both approaches can form their own "is_a" hierarchy.

However, the semantics of roles as well as their properties are still unclear as they have not been rigorously defined in previous work. In fact, developing expressive yet semantically sound type systems for object-oriented programming languages is a well-known and difficult research problem [11]. The problem is even more difficult if the concept of dynamic role playing is introduced, although the significance of the problem has been realized (e.g., see [1]).

This paper attempts to develop an expressive yet semantically sound type calculus for objects with multiple roles. The calculus is developed based on the role model used in DOOR [19], an object-role database system. While the rich role modeling constructs are introduced, the rigorous type soundness properties are preserved and expressed in a neat way by using the 'ad hoc' polymorphism [7] of overloaded functions. When we design the settings of the calculus, we try to make it more general such that it will work for different role models or similar type systems besides DOOR. We present the type-checking rules for objects with multiple roles being played and for context-dependent modeling. Furthermore, static type checking for dynamic type change is an interesting issue. To be able to check dynamic changing types statically, we need to formulate the notion of *possible types* which are the types possible to be taken by a particular object, as an object may change its types according to the program state (which needs to be determined in run-time). Therefore, dynamic type checking will produce a more definite result while it is not as good as static type checking in terms of run-time efficiency. Therefore, in this paper, both static and dynamic type checking frameworks are included and discussed individually. Finally, we seek to prove some important properties for the calculus. These include Subject Reduction, Strong Normalization and Confluence.

The organization of the rest of this paper is as follows. Section 2 outlines the DOOR data model that we use as a reference model to develop the calculus. Section 3 describes a university example that is used to illustrate the data model. In section 4, the static type checking based on the idea of possible types is presented. This primitive mechanism is complete but unsound. We will then discuss the idea of conditional types to achieve the soundness for the checking. In section 5, the type representation of an object with multiple types is defined as a type sequence. Then the calculus and type checking rules for dynamic type checking are developed. Section 6 shows some main theorems (important properties) for the calculus. Finally, section 7 concludes the paper.

2. The Reference Data Model

This section outlines the data model for DOOR (the detailed model is described in [19]) as a reference model to derive our calculus.

Object and role representation: Objects consist of object state (in terms of the values of attributes), methods, and a set of dynamically changing roles. They are referred to via their logical object ids (oid) and any oid uniquely identifies an object. The object state is encapsulated and can only be queried and updated by sending messages to the object. An object is internally organized as an acyclic graph with the root being the object itself, and all the other nodes being roles. The parent node of any node A in the graph is called the *player* (or role player) of A, and A is said to be *played-by* its player. A role is also an entry to access the object it belongs to: an object can be accessed through itself (we consider the object itself as a base role) or one of its roles, and its behavior depends on this role. On the other hand, roles encapsulate both state and behavior, but do not have a persistent, globally unique identity. A role can be itself a player and include other roles being played.

Attributes and methods: Objects are described via attributes, and all our objects are tuple-objects whose fields are the values of the object's attributes. If the attribute is single-valued, then the value is a single oid; if the attribute is set-valued, then the value is a set of oids. Since DOOR is strongly typed, a type signature needs to be assigned to each attribute in a class definition. If the signature is an object type, the attribute value must be of that type or any subtype of that type. If the signature is a role type, then the value must be an object that is playing a role of that type or of a subtype of that type. A method, invoked in the scope of an object (or a role) on a tuple of arguments, returns an answer, and, possibly, changes the state of that object (e.g., by changing the value of an attribute). As a function, each method has an arity – the number of its arguments. An attribute is regarded as a 0-ary method.

Object class and role class: Object classes have the function of organizing the persistent properties of objects into sets of related entities, while role classes organize their transient properties. The instance-of relationship between objects (or roles) and classes determines which objects (or roles) belong to which classes. The IS-A or subclass relationship, is defined between classes and is acyclic. If a class C is a subclass of another class C', then all instances of C must also belong to C'. A player-class constraint can be optionally defined in the role class to limit the possible player types of a role. If it is omitted, a player of any type is assumed. The player-class constraint is used to support the type-safe implementation of the methods in roles, as a role may invoke methods defined at its player(s). Besides the player-class constraint, other general constraints can be defined in the class-level and/or instance-level to model the fact that not every object is qualified to play a particular role. Similar to the other properties of a class, the player-class constraint of a class will be inherited by all its subclasses.

Types: The type of a class C is determined by the types of its methods, described as a signature of the form `Meth : Arg1,...,Argn` $\mapsto$ `Result`, or `Meth : Arg1,...,Argn` $\mapsto$ `{Result}`, for single-valued or set-valued methods, respectively. The signature is attached to the definition of class C, where `Argi` and `Result` are class names. When arguments that are instances of classes `Arg1,...,Argn`, respectively, are passed to the method `Meth`, the result is expected to be an instance, or a set of instances, of the class `Result`, depending on whether `Meth` is single- or set-valued, respectively. Note that there are actually $n+1$ (rather than n) arguments, where the 0th argument is not mentioned, because it is the object of class C for which the signature is defined. A method can have several signatures, each constraining the behavior of the method on different sets of arguments. When this is the case, the method is said to have a polymorphic type. The signature of a method can include role types. If a role type is included in the method signature, the corresponding object must be playing such a role and will be treated context-dependently from that perspective. Otherwise, a type violation is caused. The type of an object is more complicated and its formal description is beyond the scope of this paper. Informally, an object type consists of a static component, i.e., the type of its object class, and a dynamic component, i.e., the types of the roles being played.

Inheritance and delegation: Methods, and player-class constraints if there are any, defined in the scope of a class C are inherited by the subclasses of C through the is-a relationship. If there are different player-class constraints defined in a subclass, a most specific class will override a relatively more general one until all of them are disjoint. Inheritance is not defined for the played-by relationship. Instead, the automatic *delegation* between roles and their corresponding players is used. For example, suppose we model an employee e as a role of a person p, and *sex* is an attribute of person but not of employee. Then *sex*(e) would be a type error. We can correct this error by delegating the evaluation of *sex* to *played-by*(e) [13]. This amounts to replacing *sex(e)* by *sex(played-by(e))*.

3. Static Type Checking with Possible Types

Modern programming languages employ type checking techniques to guarantee that functions are applied only to appropriate arguments. Languages differ in the degree to which the type checking is static (performed at compile-time) or dynamic (performed at run-time). Statically typed languages, such as ML, require that function applications be proved type-safe at compile-time. This is enforced by a type inference algorithm that assigns types to program phrases. If the type inference algorithm verifies that a program cannot go wrong, the program is accepted; otherwise, the program is rejected. Static type checking eliminates the need to perform run-time type checking and detects many programming errors at compile-time. The cost of this efficiency and security is the loss of programming flexibility, because no decidable type inference system can be both sound and complete, i.e., some programs that cannot go wrong must be rejected in any statically typed language.

Dynamically typed languages, such as Lisp and Scheme, impose no type constraints on programs and, in the worst case, perform all type-checking at run-time. This permits maximum programming flexibility at the potential cost of efficiency and security. However, in an implementation of a dynamically typed language it is beneficial to perform at least some static type checking. This regains some of the benefits of statically typed languages. This is the motivation for us to include a static checking framework in dynamically typed languages for database applications. Note that the languages in which we are interested are not just dynamically typed, but also allow changing types in run-time and having multiple most-specific-types for a given object. These two features greatly increase the difficulty in formulating a static type checking framework in our language.

3.1. The Notion of Possible Types

As we cannot determine in compile-time what exact types an object will change to during its lifetime (run-time), we need to work out the possible types that an object may take by considering the played-by constraints specified in the schema. For example, we cannot tell whether a particular person object will change to student or club-chairman during its lifetime, but we can determine from the class schema that a person object o may change its type to student, club-chairman, employee, etc. during run-time. We call all these valid types of o as the set of possible types for object o. With this information, suppose that there is a function f that takes an argument of type t and now is applied to o. If t is not a supertype of a possible type of o, then a type error should be reported by the type inference mechanism. The following definitions are defined to formulate the ideas described above.

Definition 1. Let $R_o = \{r_1, r_2, \ldots, r_n\}$ be the set of all role classes such that object o of class oc can play any role instances of r_i, where $i \in [1..n]$. Then R_o is called a set of *possible types* for o. □

Since R_o is not unique, we need to define a normal form R_o^* of R_o.

Definition 2. We define R_o^* to be a set of possible types for o such that for any $r_i, r_j \in R_o^*$, $r_i \not\leq r_j$. We also define a global function PT such that PT(o) returns R_o^*. □

The implementation of *PT* can be done by tracing those played-by links in a class schema.

3.2. The Actual Run-Time Types

A role is not itself an entity but part of an object, as there is no globally unique identity for it. Therefore, types for individual roles are not of interest to us. However, we are interested in how role playing will affect the individual object types.

When a message is sent to an object from a particular role, the other roles being played by the object, but not within the same perspective, are not involved in responding to the message. This is the way that roles support the modeling of context-dependent behavior. Therefore, role acquisition and dropping do not change the run-time types of objects directly. Instead, in our settings, they will affect the possible selection of objects' run-time types. For example, a person object has to play a role as student before he/she can be treated from the student perspective.

Indeed, when sending a message to different perspectives (i.e., roles) of an object, different methods may be invoked for computation. This means that objects carry behaviors of different types when they are accessed from different roles. Therefore, the context-dependent/role-dependent access mechanism is the one that determines the run-time types of an object. The run-time type of an object is not exclusive, e.g., a TA also possesses the properties of a graduate student.

The run-time type of an object *obj* is denoted by $obj^{(A_1,A_2,..,A_n)}$ where A_1 denotes the type of the role in which the methods will be looked up first, and A_2 denotes the type of role to be looked up second and so on. In other words, A_i denotes the type of the i-th role that the methods will be looked up. The typing rules for the type sequence are defined as follows:

$$(A) \equiv A$$

$$\frac{A_i \leq A'_i, .., A_j \leq A'_j}{(A_1, .., A_i, .., A_j, .., A_n) \leq (A'_i, .., A'_j)} \; 1 \leq i \leq j \leq n$$

whereby A we denote an atomic type. We assume there is a function $RT(obj)$ that returns the run-time type of *obj*.

Since order is insisted on these multiple types of objects for method dispatching, $(\cdots)$ should be called a type sequence or an ordered set of types. Furthermore, as an object can play more than one role of the same type (i.e., it is possible to have $A_i = A_j$ where $i \neq j$), we would call $(\cdots)$ a *type sequence* instead of an ordered set. Therefore the run-time type of an object is always a sequence type if objects with multiple roles are supported. We maintain the notation $A_i \in A$ where $A = (\ldots, A_i, \ldots)$.

The order of types appearing in a type sequence is solely decided by the implementation of method lookup schemes. For example, in Fibonacci [3], both upward lookup and double lookup are supported. The former searches a method in an upward manner from a starting role while the latter looks for it in both downward and upward manners. Since an object is just a collection of roles that are organized as an acyclic graph, the notion of static type is not needed. In DOOR, in contrast, an object consists of its own state and methods (prescribed by its class, or more generally, static type) besides a set of changing roles. Therefore, under both upward and double lookup schemes in DOOR, the type sequence of an object includes its static type. This also demonstrates the idea of object specialization described by Sciore [17]. That is, objects may become more specific (specialization) by dynamically playing different roles during computation. However, they always possess the properties prescribed by their static types ("compile-time" types).

3.3. Type Checking by Set Operations

Rather than starting with a sound but uncomplete type checking mechanism, we derive a set of unsound but complete typing rules and then they will be improved/augmented with other more expensive checking techniques. The type checking rules for objects with multiple, changing types will be exactly the same as for other languages with typical type checking mechanisms, except that we are dealing with a set of types of an object instead of having a fixed type for each object. To save space, we discuss only the rules for some interesting constructs, which include basic type, substitution, type change, and function application.

$$\frac{\vdash o : PT(o) \qquad \tau \in PT(o)}{\vdash o : \tau}$$

$$\frac{\vdash o : PT(o) \qquad \tau \in PT(o) \qquad \vdash \tau \leq \tau'}{\vdash o : \tau'}$$

$$\frac{\vdash add : R -> R \cup \{\tau'\} \qquad R \cap R' \neq \phi \qquad \vdash o : R'}{\vdash add(o) : R \cup \{\tau'\}}$$

$$\frac{\vdash sub : R -> R - \{\tau'\} \qquad R \cap R' \neq \phi \qquad \vdash o : R'}{\vdash sub(o) : R - \{\tau'\}}$$

$$\frac{\vdash m : \tau -> \tau' \qquad \tau \in PT(o) \qquad \vdash o : PT(o)}{\vdash m(o) : \tau'}$$

The above type checking rules are obviously complete but unsound. For example, the first rule means that every type in the set of possible types of o is itself a type of o. This is definitely not always true. For instance, suppose that student is a possible type of person. Certainly, not every person is a student. However, these typing rules will help us to filter out the obviously incorrect answers in no time. For example, if one tries to change an instance of person to a vehicle, a type error can be immediately produced. Similarly, if a function which expects a vehicle as an argument is applied to an instance of person, a type error should be reported without going through further checking/analysis.

3.4. Soundness Improvement with Conditional Types

With conditional types [2], the type of an expression e that may result in an object o can be constrained using information about the results of run-time tests in the context surrounding e. For example, in an expression

$$\textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3$$

conditional types can express that e_2 is evaluated only in environments where e_1 is true, and e_3 is evaluated only in environments where e_1 is false. This can be done by the idea of control-flow analysis. Control-flow information in type inference is crucial to computing accurate type information in dynamically typed programs like the ones mentioned in this paper. For instance, by keeping track of those statements for changing types (e.g. *add* and *sub*), we can determine a sound type checking mechanism. Therefore, we propose a two-pass static type checking. The first pass is based on the idea of possible types while the second is based on the control-flow analysis. It is no doubt that the second pass is more expensive than the first pass.

4. Dynamic Type Checking with Overloaded Functions

4.1. Overloading, Subtyping, and the Calculus

This subsection develops the typed $\lambda\mathcal{R}_{\&}$-calculus, which is used to provide a theoretical foundation for objects with roles. The calculus is inspired by the calculus for *function overloading* [8]. The code of an overloaded function is formed by several branches of code. The branch to execute is chosen, when the function is applied, according to a particular selection rule which depends on the type of the argument. The crucial feature of the present approach is that a subtyping relation is defined among types, such that the type of a term generally decreases during computation, and this fact induces a distinction between the 'compile-time' type and the 'run-time' type of a term. Castagna, Ghelli, and Longo [8] studied the case of overloaded functions where the branch selection depends on the run-time type of the argument, so that overloading cannot be eliminated by a static analysis of code, but is an essential feature to be dealt with during computation. Using their result, we can develop the method selection rules for objects with *multiple* and *changing* types.

The motivation of overloaded functions comes from considering overloading as a key feature of object-oriented programming, when methods are viewed as 'global' functions. In object-oriented languages the computation evolves on objects. When an object receives a message it invokes the method associated to that message. The association between methods and messages is described by the class, as an array, the object belongs to. Thus objects are implemented as a tuple of internal state and class name (as well as the roles being played, if roles are introduced to the languages). This implementation has been extensively studied and corresponds to the 'objects as records' analogy of [6]. Another way to implement message-passing is to consider messages as names of overloaded functions: according to the class (or more generally, the type) of the object the message is passed to, a different method is chosen (this approach is used in CLOS [10]). By this, in a sense, we inverse the previous situation: instead of passing messages to objects we now pass objects to messages. Objects are still implemented by a tuple of internal state and class name (and roles being played), and become in this way arguments of overloaded functions.

Indeed in the first approach objects carry methods with them; thus the types of the objects contain also the functionality of the value. This causes some problems and requires an excessive use of recursion. On the contrary, in the overloading approach, the type of an object is no longer blurred by functional types. The functionality is fully expressed by methods as global, overloaded functions. Of course other problems arise, especially in the modeling of the encapsulation of the state, though they do not seem overwhelming. On the other hand, the full expressiveness of records is recovered, as record types and values are derivable notions in our approach.

It is clear that overloaded functions express computations which depend on input types. This will save us a lot of effort and complexity to consider different set of methods encapsulated by different changing roles of an object, as different codes of overloaded functions may be applied on the basis of input types.

4.2. Types

We extend the traditional lambda calculus with types to be defined in this subsection. We first define a set of *Pretypes* and then among them we will select those that satisfy the conditions to constitute the types.

$$\begin{aligned} \mathit{PreTypes} \ni V \ ::= \ & A \mid (A_1, \ldots, A_n) \mid V \to V \mid \\ & \{V_1 \to V_1', \ldots, V_n \to V_n'\} \end{aligned}$$

where A is an atomic type (including ϕ). Subtyping relation on *Pretypes* is obtained by adding the rules of transitive and reflexive closure to the following:

$$\frac{U_2 \leq U_1 \quad V_1 \leq V_2}{U_1 \to V_1 \ \leq \ U_2 \to V_2}$$

$$\frac{\forall i \in I,\ \exists j \in J\ U_i'' \leq U_j' \text{ and } V_j' \leq V_i''}{\{U_j' \to V_j'\}_{j \in J} \ \leq \ \{U_i'' \to V_i''\}_{i \in I}}$$

as well as the ones for type sequence above. Intuitively if we consider two overloaded types U and V as a set of functional types then the second rule states that $U \leq V$ if and only if for every type in V there is one in U smaller than it. We can now define *Types*:

1. $A \in \mathit{Types}$

2. $(A_1, \ldots, A_n) \in \mathit{Types}$
3. if $V_1, V_2 \in \mathit{Types}$ then $V_1 \to V_2 \in \mathit{Types}$
4. if $\forall i, j \in I$
   (a) $(U_i, V_i \in \mathit{Types})$ and
   (b) $(U_i \leq U_j \Rightarrow V_i \leq V_j)$ and
   (c) $(U_i \Downarrow U_j \Rightarrow$ there is a unique $h \in I$ such that $U_h = \inf\{U_i, U_j\})$

   then $\{U_i \to V_i\}_{i \in I} \in \mathit{Types}$

where $U_i \Downarrow U_j$ means that U_i and U_j are downward compatible, i.e., they have a common lower bound. An overloaded type is inhabited by functions made out of different pieces of code. When an overloaded function is applied to an argument, a choice is made of the code that will be actually used in the computation. The choice is based on the type of the argument and the condition of 4(c) assures its uniqueness.

4.3. Terms

Informally, terms correspond to terms of the classical lambda calculus plus an operation which concatenates two different branches and forms an overloaded term. Since we want the branches of an overloaded function to be ordered, then we construct them as customary with lists, i.e., we start by an empty overloaded function and add branches, concatenated by ampersands. We also distinguish the usual application $M \cdot M$ of lambda calculus from the application of an overloaded function $M \bullet M$ since they constitute two completely different mechanisms: indeed to the former is associated a notion of variable substitution while in the latter there is the notion of selection of a branch. This is stressed also by the proof-theoretical viewpoint where these constructors correspond to two different elimination rules. Finally, a further difference, specified in the reduction rules, is that overloaded application is associated to call by value, which is not needed by the ordinary application. For same reason we must distinguish between the type $U \to V$ and the overloaded function type with just one branch $\{U \to V\}$.

$$\begin{aligned}\mathit{Terms} \ni M ::= {} & x^V \mid \lambda x^A.M \mid M \cdot M \mid \\ & \varepsilon \mid M \&^V M \mid M \bullet M\end{aligned}$$

4.4. Type Checking Rules

The rules to type-check the calculus are shown in Figure 1. The *subsumption rule* is not used in the type rules, however, the system enjoys the subsumption property, i.e., for any $U \leq V$ and for any context $C[]$ and terms $M : U$ and $N : V$, if $C[M]$ is well-typed then $C[N]$ is well-typed too. This means that our system could be presented using the subsumption rule. Note that with the subsumption rule the runtime type of a term (used only in the reduction rules, to perform branch selection), should be defined as the minimum type of a closed normal term.

$[taut_\varepsilon]$ $$\vdash \varepsilon : \phi$$

$[taut_:]$ $$\vdash x^V : V$$

$[intro_\to]$ $$\frac{\vdash M : V}{\vdash \lambda x^A.M : A \to V}$$

$[elim_{\to,\leq}]$ $$\frac{\vdash M : U \to V \quad \vdash N : W \leq U}{\vdash M \cdot N : V}$$

$[intro_\&]$ $$\frac{\begin{array}{c}\vdash M : W_1 \leq \{U_i \to V_i\}_{i \leq (n-1)} \\ \vdash N : W_2 \leq U_n \to V_n\end{array}}{\vdash (M \&^{\{U_i \to V_i\}_{i \leq n}} N) : \{U_i \to V_i\}_{i \leq n}}$$

$[elim_\&]$ $$\frac{\begin{array}{c}\vdash N : (A_1, .., A_n) \\ \vdash M : \{U_i \to V_i\}_{i \in I} \\ U_j = \mathtt{min}_{k \text{ s.t. } U_j \neq \phi} \mathtt{min}_{i \in I} \{U_i | A_k \leq U_i\}\end{array}}{\vdash M \bullet N : V_j}$$

Figure 1: Type checking rules for the calculus.

5. Important Properties

5.1. Reduction

For simplicity, we consider the types of overloaded functions as ordered sets. The order corresponds to the order in which branches appear, i.e., in which they are 'constructed' according to the rules. Also, we will allow a reduction of the application of an overloaded function only when its argument is in normal form. This is a crucial point, because if the argument of an overloaded function is reduced, its type may change. Therefore, a different branch of the overloaded function might be chosen. As a matter of fact, in object-oriented languages one can send messages only to objects in normal form (see [8] for further description). In summary, we define the reduction relation $\rhd$ as follows:

(β): $(\lambda x^A.M)N \ \rhd\ M[x^A := N]$

$(\beta_{\mathcal{R}_\&})$: If $N : (A_1, \ldots, A_m)$ is closed and in normal form and $U_j = \min_k \min_i \{U_i | A_k \leq U_i\} \neq \phi$ then

$$((M_1 \&^{\{U_i \to V_i\}_{i=1 \ldots n}} M_2) \bullet N) \rhd \begin{cases} M_1 \bullet N & \text{for } j < n \\ M_2 \cdot N & \text{for } j = n \end{cases}$$

$(\mathit{context})$: If $M_1 \rhd M_2$ then

$$\begin{array}{ccc} (M_1 N) & \rhd & (M_2 N) \\ (N M_1) & \rhd & (N M_2) \\ (\lambda x^A.M_1) & \rhd & (\lambda x^A.M_2) \\ (M_1 \& N) & \rhd & (M_2 \& N) \\ (N \& M_1) & \rhd & (N \& M_2) \end{array}$$

In $(\beta_{\mathcal{R}_\&})$, $U_j = \min_k \min_i \{U_i | A_k \leq U_i\} \neq \phi$ is used to find the first type in the type sequence (min_k) of the object such that the overloaded function $(\mathit{min}_i U_i)$ can be found. The reasons of the two restrictions in the $(\beta_{\mathcal{R}_\&})$ is that two operations may change the type of a term: namely, reduction and substitution.

Since we want the type of the argument of an overloaded function to be fixed, we require that it is in normal form in order to avoid reductions and that it is closed in order to avoid substitutions. The intuitive operational meaning of $(\beta_{\mathcal{R}_{\&}})$ is easily understood when looking at the simple case, i.e., when there are many branches as arrows in the overloaded type. In this case, under the assumptions in the rule, one has

$$(M_1 \& \ldots \& M_n)(N) \ \rhd^* \ M_j N$$

The nested formalization above of $(\beta_{\mathcal{R}_{\&}})$ is needed as M_1 may be an application $P_1 Q_1$, i.e., the 'external operation' in M_1 is application, instead of an &.

Theorem 1. Every well-typed $\lambda\mathcal{R}_{\&}$-term possesses a unique type. □

5.2. Main Theorems

In this subsection we present the main results for our calculus. The results closely follow the idea and settings in [8]. We start with a generalization of the subject reduction theorem which states that if a term is typeable then it can be reduced only to typeable terms and these terms have a type lesser than or equal to the type of the redex.

Theorem 2. (Generalized Subject Reduction) Let $M : U$. If $M \rhd^* N$ then $N : U'$, where $U' \leq U$. □

This theorem is important because it states that the computation is well-behaved with respect to types. Next we have the Strong Normalization theorem. As is well-known, strong normalization cannot be proved by induction on terms, since β-reduction potentially increases the size of the reduced term. For this reason we introduce, along the lines of [14], a different notion of induction on typed term. This notion is shaped over reduction, so that some reduction related properties, like strong normalization or confluence, can be easily proved to be typed-inductive. We entirely omit the proof and just note that the main lemma for it, which proves that every typed-inductive property is satisfied by any typed term.

Theorem 3. (Strong Normalization) Terms strongly normalize. □

Now we can prove the (syntactical) consistency of the calculus.

Theorem 4. (Church-Rosser) If $M \rhd P$ and $M \rhd Q$ then there exists N such that $P \rhd^* N$ and $Q \rhd^* N$. □

The proof of this theorem is technically the easiest of the three since it is not difficult to show that the calculus is weakly Church-Rosser. Then, by the Newmann's lemma, one directly derives the Church-Rosser property for it. This theorem is important since it assures that no matter how the calculus is implemented it always returns the same result. Thus it behaves in a deterministic way.

6. Summary

As the idea of objects with multiple, changing types, or called 'objects with roles' is getting popular in the context of programming languages as well as database systems (especially object evolution in object-oriented databases), in this paper, we have provided a theoretical foundation for objects with multiple, changing types (in terms of roles). This work is entirely novel. We have formulated a calculus to analyze the semantics and type issues of roles. Overall, the paper can be concluded as follows:

- The setting of our calculus is general and fits various role models with different message passing semantics. Moreover, it can be easily adjusted to model the different subtle aspects of different models. For example, the compile type (or static type) of an object can be omitted or assumed to be ϕ to model that an object is simply a collection of roles in Fibonacci [3].
- We point out that the different (multiple) types of an object are actually determined by the role in which the object is accessed (i.e., context-dependent access). Therefore, role update operations (such as acquiring roles or dropping roles) are not directly related to the object type determination, though they affect what role is available to be selected as an object entry point. As a result, the determination of types can be separated from the dynamic role updates.
- Using the idea of overloaded functions for constructing the calculus, both objects and roles will contain only their internal states and roles being played. This helps in simplifying the calculus as methods are pulled out of classes. Moreover, method selection is used, instead of message passing, to implement delegation among the roles being played by an object. This provides at least an alternative to describe delegation among roles, apart from those informal descriptions used in [3, 19].
- We have shown the rules to type check an object with multiple roles (or multiple types). We have also proved some important properties of the calculus for roles, in terms of the proofs for Subject Reduction, Strong Normalization and Confluence theorems.
- Both static and dynamic checking frameworks are described. It is unlikely to be able to check types in compile-time without formulating a set of possible types which are possible to be taken by a particular object, as an object may change its types according to the program state (which needs to be determined in run-time). Therefore, dynamic type checking will produce a more definite result while it is not as good as static type checking in terms of run-time efficiency.

More work is needed to be done on objects with roles, or the context of 'types evolving during computations'. For instance, our ongoing work is to investigate the expressive power of object with multiple roles. In fact, given the reference model described in this paper, we can show that object with roles is at least as powerful as parametric type classes (e.g., see [9]), as the former can model the latter but not vice versa. For example, a method of a role class may have an argument type or return type which depends on its player type. Along this line, this idea can generate the modeling of parametric type classes with roles.

References

[1] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of databases.* Addison-Wesley, 1995.

[2] A. Aiken, E.L. Wimmers, and T.K. Lakshman. Soft typing with conditional types. In *Proceedings of ACM POPL'94*, pages 163–173, 1994.

[3] A. Albano, G. Ghelli, and R. Orsini. Fibonacci: A Programming Language for Object Databases. *VLDB Journal*, 4(3):403–444, 1995.

[4] C.W. Bachman and M. Daya. The role concept in data models. In *Proceedings of the Third International Conference on Very Large Databases*, pages 464–476, 1977.

[5] E. Bertino and G. Guerrini. Objects with Multiple Most Specific Classes. In *ECOOP'95 - Object-Oriented Programming*, pages 102–126. Springer LNCS952, 1995.

[6] L. Cardelli. A Semantics of Multiple Inheritance. *Information and Computation*, 76:138–164, 1988.

[7] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[8] G. Castagna, G. Ghelli, and G. Longo. A Calculus for Overloaded Functions with Subtyping. In *ACM Conference on LISP and Functional Programming*, June 1992.

[9] K. Chen, P. Hudak, and M. Odersky. Parametric Type Classes. In *ACM Conference on LISP and Functional Programming*, June 1992.

[10] L.G. DeMichiel and R.P. Gabriel. Common Lisp Object System Overview. In *Proceedings of ECOOP'87*, 1987.

[11] J. Eifrig, V. Smith, V. Trifonov, and A. Zwarico. An interpretation of typed oop in a language with state. *Lisp and Symbolic Computation*, 8(4), 1995.

[12] G. Gottlob, M. Schrefl, and B. Rock. Extending Object-Oriented Systems with Roles. *ACM Transactions on Information Systems*, 14(3):268–296, July 1996.

[13] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In N. Meyrowitz, editor, *Object-Oriented Programming: Systems, Languages and Applications*, pages 214–223, October 1986.

[14] J.C. Mitchell. A type inference approach to reduction properties and semantics of polymorphic expressions. In *ACM Conference on LISP and Functional Programming*, 1986.

[15] B. Pernici. Objects with roles. In *ACM Conference on Office Information Systems*, pages 205–215, Cambridge, Mass., 1990.

[16] J. Richardson and P. Schwartz. Aspects: Extending objects to support multiple, independent roles. In *ACM-SIGMOD International Conference on Management of Data*, pages 298–307, Denver, Colorado, May 1991. ACM SIGMOD Record, Vol. 20.

[17] E. Sciore. Object Specialization. *ACM Transactions on Information Systems*, 7(2):103–122, April 1989.

[18] L.A. Stein and S.B. Zdonik. Clovers: The Dynamic Behavior of Type and Instances. Technical Report CS-89-42, Brown University, November 1989.

[19] R.K. Wong, H.L. Chau, and F.H. Lochovsky. A Data Model and Semantics of Objects with Dynamic Roles. In *IEEE International Conference on Data Engineering*, pages 402–411, April 1997.