Foundations for Extensible Objects with Roles
Extended Abstract

Giorgio Ghelli and Debora Palmerini

Abstract

Object-oriented database systems are an emerging, promising technology, underpinned by the integration of ideas from object-oriented languages along with the specific needs of database applications.

The fundamental reason for using such systems is that any real-world entity can be modelled with one object which matches its structure and behavior. To this end, the standard notion of object has to be augmented so that it can model the fact that an entity may acquire new pieces of structure and behavior during its existence, without changing its identity. To allow this extensibility in a statically typed system, a notion of context-dependent behavior ("role playing") has to be added to the basic features of object-oriented languages. This feature is also a useful modeling device.

Languages with role mechanisms have already been proposed. However, their design is full of choices which cannot be easily justified. A strong foundation for the object-with-roles notion would be extremely helpful to justify these choices and to understand, and prove, the properties of such a mechanism. In this paper we describe such a foundation, building on the object model proposed by Abadi and Cardelli.

1 Introduction

In the database field, the object-oriented data model attracts much attention because of its ability to faithfully represent real world entities. However, database applications need an operation, which we call *object extension*, which is not allowed in the standard object-oriented model. Object extension is the operation which allows an object, created in a class C, to become an instance of a subclass S too, without changing its *identity*.

The problematic aspect of extension can be better explained by an example. Consider an object type *Person* with two subtypes, *Student* and *Employee*, which both introduce an *IdCode* field, with a different meaning and even a different type. Extension allows one to build a student *John* with IdCode 100 and then to extend it to be also an employee with IdCode "I1". It is not clear, now, how *John* should answer an *IdCode* message.

We call "incompatible" such an extension that adds an already present field, with a non compatible type. In the FOOL ("Foundations of O-O Languages") field it is usually assumed that such an incompatible extension should be always forbidden.

Another alternative, studied in the field of database languagues ([GSR96], [BG95], [Run92], [AGO95]...), is to give *John* a context dependent behavior: in different contexts *John* plays either the *Student* or the *Employee* role, and answers the *IdCode* message in a role-dependent way. The idea of objects with multiple roles, whose behavior depends on the role played, is also a useful modeling device, which combines the flexibility given by method overriding with the ability to access different methods in different situations.

In the Pisa University database group we have defined and developed a database programming language, *Fibonacci*, which embodies these ideas ([AGO95]). During this process, we had to make some design choices, and to adopt some typing rules, whose real meaning was not totally clear to us. Our understanding of the object with roles mechanism was not complete, and this paper tries to fill this gap.

The paper is structured as follows. In Section 2 we recall Abadi-Cardelli ς-calculus, which is the basis of our proposal. In Section 3 we give an informal introduction to our calculus. The calculus, in its basic form, is formally introduced in Section 4. Section 5 enriches the calculus with an inheritance mechanism. Section 6 describes an important technical point, the internal structure of the set of role-tags. Section 7 discusses some related works. Section 8 draws some conclusions.

2 The ς-calculus

Our model is defined as an extension of Abadi-Cardelli ς-calculus. In that calculus, an object is simply a method suite, where each method has a special "self-variable", bound by the ς binder. Three operations are defined on objects: construction $[l_i = \varsigma(x_i : A)b_i^{i\in I}]$, method selection $a.l$, and method update $a.l \leftarrow \varsigma(x_i : A)b$. Method selection returns the body of the selected method and substitutes the "self-variable" with the object; method update updates the body of a method. The syntax of the calculus is defined below; type rules and operational semantics can be found in [AC96].

$$\begin{array}{llcl}
\textbf{Types} & A, B & ::= & K \mid [l_i : B_i{}^{i\in I}] \\
\textbf{Terms} & a, b, o & ::= & x \mid k \mid [l_i = \varsigma(x_i : A)b_i{}^{i\in I}] \mid o.l \mid o.l \leftarrow \varsigma(x_i : A)\ b
\end{array}$$

3 An overview of the calculus

3.1 The Fibonacci model

Our role model is an abstract version of the Fibonacci model, which is better explained by an example. The following piece of Fibonacci code defines three object types, then builds a student, and extends it to an employee.

```
Let Person   = IsA NewObject  With Name: String; End;
Let Student  = IsA Person     With IdCode: Int; End;
Let Employee = IsA Person     With IdCode: String; End;

let john           = object Person
                       methods Name = "John" end;
let johnAsStudent  = extend john to Student
                       methods IdCode = 100 end;
let johnAsEmployee = extend johAsStudent to Employee
                       methods IdCode = "I1" end;
```

According to Fibonacci "arrows and boxes" informal model, the construction and extension operations above build an object with an internal structure of three *roles*, one for each different object type owned by the object. Each of the three identifiers `john...` denotes a different role of the same object, as depicted in Figure 1.

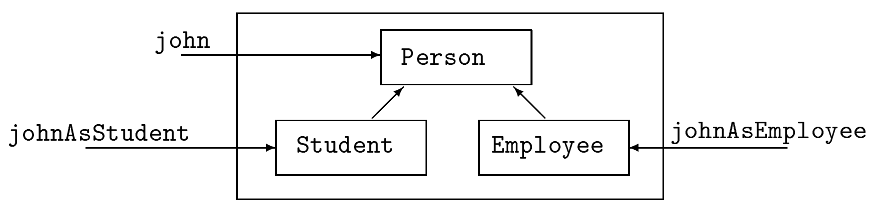

Figure 1: The internal structure of an object with roles

A message can be sent to a role either with the $o!l$ or with the $o.l$ notation. In the first case, the method is looked for in the receiving role and in its ancestors (*upward lookup*). In the second case, the method is first looked for in the role descendants, starting from last one acquired; if no descendant has a method for the message, upward lookup is performed (*double lookup*). If the method is found during downward lookup, then an instance of *self* in that method denotes the role where the method has been found; otherwise, *self* denotes the role which received the message. If extension is never used, then every object is always accessed from its bottom role, hence the double and upward lookup coincide, and both coincide with the standard Smalltalk rule, including the semantics of *self*. In this paper we will only address upward lookup, but double lookup does not introduce additional problems.

3.2 The abstract model

The essential features of the Fibonacci model that we would like to represent are:

1. classical smalltalk-like objects are a special case of objects with roles (other

proposals support roles at the expense of other features, such as dynamic binding);

2. a "role expression" denotes one specific role of an object; messages are sent to roles, and method lookup depends on the receiving role (in other approaches, messages are sent to objects, and it is the context, namely the static type of the receiver, which influences method lookup, as in [BG95]);

3. object types (more precisely, role types) are generative: the `Isa` operator generates a brand new type whenever it is invoked; for example, *Employee* and *Student* would be two different types even if *IdCode* were an integer in both cases;

4. an object is not allowed to acquire the same role type twice: extending a student to the type *Student* is not allowed.

Features (1) and (2) are fundamental and easily defendable design choices, while (3), hence (4), are more questionable.

In the type theoretic field, we usually prefer to deal with non-generative object types, mainly because generative types, which may be seen as a limited form of dependent types, have bad interactions with other constructs, such as modules and polymorphism. In the database field, on the other side, we prefer generative types because a *Person* models a class of entities which "happen" to have a certain interface, but the "identity" of the type, and its position in the type hierarchy, cannot be simply identified with its interface.

We chose here to model generative types to have a more faithful model for Fibonacci, and also because we believe that generative object types is an important notion which needs better foundations. The system we present here models generative types in a non-generative context by exploiting the idea of "role-tags"; in Section 6 we spend some more words on this idea.

Finally, we adopt here constraint (4) because it is found in Fibonacci, but it may be dropped without major consequences.

For the sake of simplicity, we will only model *upward lookup* and, to keep with the tradition, we will use the standard notation $o.l$ instead of the Fibonacci $o!l$; modelling *double lookup* too would not add much to the work.

To model objects with roles we proceed as follows. Since methods are selected on the basis of a message and a role, we index methods in an object with a role name–message pair. The "role name" is chosen from an infinite set $\mathcal{R}$ of role-tags. Then, since an "object expression" actually denotes one specific role of an object, we transform objects into role-tag–object pairs. Hence, an object-with-role playing the role R is now represented as follows, where the *current role* R belongs to $\{R_i\}^{i \in I}$:

$$\langle R, [R_i, l_i = \varsigma(x_i : A)\ b_i{}^{i \in I}] \rangle$$

Every role-tag R, R_i comes from an arbitrary partially ordered set $\mathcal{R}$. Our theory is independent of the chosen $\mathcal{R}$, hence we can assume that whichever object type hierarchy we are interested in, this hierarchy is chosen as $\mathcal{R}$. For example the previous example can be modelled by taking

$$\mathcal{R} = \langle\{\text{Pers, Stud, Emp}\}, \text{Ord}\rangle$$

where *Ord* is order generated by $Stud \leq Pers$, $Emp \leq Pers$. We can do better, however, and define a special set $\mathcal{R}$ where every finite object type hierarchy can be "faithfully" embedded. This construction is explained in Section 6 only, since our particular notion of "faithfulness" can be better understood after the calculus has been presented.

This calculus, with the reduction rules that we will define, allows one to model the *john*, *johnAsStudent*, *johnAsEmployee* values which are produced by the previous Fibonacci operations as follows.

$$\text{john} = \langle Pers, [Pers, Name = \varsigma()\ \text{"John"}]\rangle$$

$$\begin{aligned}\text{johnAsStudent} = \langle Stud, [&Pers, Name = \varsigma()\ \text{"John"};\\ &Stud, Name = \varsigma()\ \text{"John"};\ Stud, IdCode = \varsigma()\ 100]\rangle\end{aligned}$$

$$\begin{aligned}\text{johnAsEmployee} = \langle Emp, [&Pers, Name = \varsigma()\ \text{"John"};\\ &Stud, Name = \varsigma()\ \text{"John"};\ Stud, IdCode = \varsigma()\ 100;\\ &Emp, Name = \varsigma()\ \text{"John"};\ Emp, IdCode = \varsigma()\ \text{"I1"}]\rangle\end{aligned}$$

We will define a side-effect free calculus, as is common in the type-theoretic field, to be able to study the essential features avoiding some unnecessary complications. More precisely, though the notion of 'object identity' is not modeled in side-effect calculi, our study will nevertheless face the type-theoretic problems which are posed by identity preserving update, while avoiding to deal with stores and locations. This presence of the typing problems of imperative object-oriented languages in the functional setting is a well-known phenomenon, which is explained by the presence of *self*, combined with the requirement that methods which have been type-checked before functional update of the object should not need to be checked once more after the update. Informally, an updated object is referenced both by the instances of *self* in the methods checked before update and by those in the methods added by the update operation. This form of sharing, althought limited, already presents the same type-theoretics challenges that arise in the imperative setting because of the full sharing allowed by the presence of updatable locations. The extension of this calculus to an imperative one is relatively straightforward.

Since this basic calculus is modelled over the ς-calculus, it has no inheritance operator, and inheritance can be represented using the same techniques as in [AC96]. However, the example above shows that here inheritance is more important than in usual object calculi: in object calculi inheritance is used to avoid code replication in the definition of different objects (or classes), while here we have to deal with code replication inside one single object. For this reason, we will also study a version of the role calculus with inheritance, by giving a translation onto the basic role calculus. However we start with the inheritance-free calculus because we are looking for the simplest calculus where the notion of roles can be studied.

4 The basic calculus

4.1 The syntax

The role calculus extends the ς-calculus by indexing methods with role-tag–label pairs and by pairing each object with a "current role". We also extend the calculus with some role-related operations:

1. object extension: this operation adds a new set of methods to an object; the role-tag–label pairs of the new methods are required not to appear in the object:
$$o + [R, l_i = \varsigma(x_i : A)\ b_i\ {}^{i \in I}]$$
2. role navigation: the operation o **as** R sets the current role-tag of o to R;
3. role checking: the operation o **is** R tests the current role-tag of o;
4. dynamic type cast: the operation **check**$(a : A)$ casts a to the object type A, and fails if this is not sound; this operation is not part of the kernel of every role calculus, but is very useful in practice.

The syntax of the calculus is thus defined as follows, where R and R_i range over $\mathcal{R}$.

Types	A, B	$::=$	$K \mid \langle R, [R_i, l_i : B_i\ {}^{i \in I}]\rangle \mid \langle R, [R_i, l_i : B_i\ {}^{i \in I}]\rangle^+$
Terms	a, b, o	$::=$	$x \mid k \mid \langle R, [R_i, l_i = \varsigma(x_i : A)\ b_i\ {}^{i \in I}]\rangle$
			$\mid o.l \mid o.l \leftarrow \varsigma(x : A)\ b$
			$\mid o + [R, l_i = \varsigma(x_i : A)\ b_i\ {}^{i \in I}]$
			$\mid o$ **as** $R \mid o$ **is** $R \mid$ **check**$(a : A)$
Environments	E	$::=$	$() \mid E, x : A$
Judgements	J	$::=$	$E \vdash\ \diamond \mid\ \vdash A\ \diamond \mid E \vdash a : A \mid\ \vdash A \leq B$

Note that in the object construction and object extension operations A does not depend on i because all methods must declare the same type for their *self* parameter x_i.

4.2 Typing and subtyping

As in [AC96], the type of an object describes the structure of the object itself, hence its syntax is $\langle R, [R_i, l_i : B_i\ {}^{i \in I}]\rangle$, and the type rules for object formation, method extraction, method update, and subsumption, are essentially as in [AC96].

We would like to have a non trivial subtype relation, including at least width subtyping (more fields in a subtype), as in Abadi-Cardelli calculus. However, we also have to type the object extension operation, with the constraint that a role-tag–message pair cannot be acquired twice. Subsumption combined with width subtyping implies that from the type of a record we can only read the presence of a field but not its absence, which makes it impossible to check the constraint on object extension. This is a classical problem, which we solve in the simplest way, by defining both a strict and a weak object type. The strict type $\langle R, [R_i, l_i : B_i{}^{i \in I}] \rangle$ describes the exact structure of an object, hence only trivial subtyping is defined on strict types (rule [StrictSub-Form] below), and strict types are used to type the extension operation (rule [Ext]). The weak object type $\langle R, [R_i, l_i : B_i{}^{i \in I}] \rangle^+$ lists some messages which are guaranteed to be answered by the object, hence width subtyping applies, and weak types are used to type method extraction (rule [Meth]). We use strict types to type-check method updates too (rule [Upd]), hence we gain deep subtyping on weak types (rule [WeakSub]). Strict types can be promoted to the correspondent weak type (rule [StrictWeakSub]). Hereafter, unqualified "object type" stands for the weak version. The use of strict and weak types to type update and query operations respectively was first proposed in [Ghe90], and developed independently, for object update, in [FM96]; it is also strictly connected with the idea of "row variables".

Weak object subtyping also allows the current role to be promoted to a super-role. This happens because we want, for example, to be able to write a function to print the name of a person as follows, and then to apply it to students and employees.

```
let type Person = <Pers,[Pers,Name:String]>+;
let printName = fun(x:Person)  printString(x.Name);
```

However, role promotion create a soundness problem. It would not be sound to pass an object o whose strict type is $\langle \mathit{Stud}, [\mathit{Pers}, \mathit{Name} : \mathit{string}] \rangle$ to the function above, since `x.Name` would look for a *Stud,Name* method, but o is not able to answer the *Name* method in its *Student* role (we have no inheritance here); however, the type of o *is* a subtype of $\langle \mathit{Pers}, [\mathit{Pers}, \mathit{Name} : \mathit{string}] \rangle$. We solve this problem by considering such an object as ill formed: if a student can answer a method m as a person, it must be able to answer m as a student too. This "downward closure" condition is formalized in the fifth premise of rule [StrictSub-Form], and will come (almost) for free in the version with inheritance. The premise is better read as: for every method R_j, l_j and for every role $R_i \leq R_j$ which appears in some other method, there is a method R_h, l_h which answers the message l_j for the role R_i (i.e., $\langle R_h, l_h \rangle = \langle R_i, l_j \rangle$).

A problem would also arise if we allowed an object with strict type $\langle \mathit{Stud}, [\mathit{Stud}, \mathit{Name} : \mathit{int};\ \mathit{Pers}, \mathit{Name} : \mathit{string}] \rangle$ to be passed to the same function. In this case, the *Stud*, *Name* method answers the call `x.name` which has been typed with respect to the *Pers*, *Name* method, hence the type of the first method must be a subtype of the type of the second. This "covariance" condition is captured by the fourth premise of rule [StrictSub-Form]. Notice that this covariance is orthogonal to the deep subtyping question, but is strictly related to the same condition we find in the λ-& calculus of overloaded functions with late binding [Ghe91, CGL95]. For reasons of space we cannot discuss this point any further.

We are now ready to present the most important typing rules of our system. We only omit the obvious rules for variables and constants. We define good formation $\vdash A \diamondsuit$ as $\vdash A \leq A$.

Subtyping and type formation

$$\frac{\begin{array}{ll}(1) & \forall i \neq j.\ \langle R_i, l_i\rangle \neq \langle R_j, l_j\rangle \\ (2) & \forall i \in I.\ \vdash B_i \ \Diamond \\ (3) & R \in [R_i, l_i = \varsigma(x_i : A)\ b_i{}^{i\in I}] \\ (4) & \forall i,j \in I.\ R_i \leq R_j, l_i = l_j \ \Rightarrow \vdash B_i \leq B_j \\ (5) & \forall i,j \in I.\ R_i \leq R_j \ \Rightarrow\ \exists h \in I.\ \langle R_h, l_h\rangle = \langle R_i, l_j\rangle\end{array}}{\vdash \langle R, [R_i, l_i : B_i{}^{i\in I}]\rangle \leq \langle R, [R_i, l_i : B_i{}^{i\in I}]\rangle} \quad \text{[StrictSub-Form]}$$

$$\frac{\begin{array}{l}\forall i' \in I'.\ \vdash B'_{i'} \ \Diamond \\ R' \leq R \quad \forall i \in I.\ \exists i' \in I'.\ R'_{i'} = R_i, l_{i'} = l_i, \vdash B'_{i'} \leq B_i\end{array}}{\vdash \langle R', [R'_i, l'_i : B'_i{}^{i\in I'}]\rangle^+ \leq \langle R, [R_i, l_i : B_i{}^{i\in I}]\rangle^+} \quad \text{[WeakSub]}$$

$$\frac{\vdash \langle R, [R_i, l_i : B_i{}^{i\in I}]\rangle \ \Diamond}{\vdash \langle R, [R_i, l_i : B_i{}^{i\in I}]\rangle \leq \langle R, [R_i, l_i : B_i{}^{i\in I}]\rangle^+} \quad \text{[StrictWeakSub]}$$

Term formation

$$\frac{\begin{array}{c}\text{let } A = \langle R, [R_i, l_i : B_i{}^{i\in I}]\rangle \\ \forall i \in I.\ E, x_i : A^+ \vdash b_j : B_j\end{array}}{E \vdash \langle R, [R_i, l_i = \varsigma(x_i : A)\ b_i{}^{i\in I}]\rangle : A} \quad \text{[ObjIntro]}$$

$$\frac{\begin{array}{l}\text{let } A = \langle R, [R_i, l_i : B_i{}^{i\in I}; R, l_j : B_j{}^{j\in J}]\rangle \\ E \vdash a : \langle R', [R_i, l_i : B_i{}^{i\in I}]\rangle \quad \vdash A \ \Diamond \\ \forall j \in J.\ E, x_j : A^+ \vdash b_j : B_j\end{array}}{E \vdash a + [R, l_j = \varsigma(x_j : A^+)\ b_j{}^{j\in J}] : A} \quad \text{[Ext]}$$

$$\frac{\begin{array}{l}E \vdash a : A = \langle R, [R_i, l_i : B_i{}^{i\in I}]\rangle \\ \exists h \in I.\ \langle R_h, l_h\rangle = \langle R, l\rangle \quad E, x : A^+ \vdash b : B_h\end{array}}{E \vdash a.l \leftarrow \varsigma(x : A^+)\ b : A} \quad \text{[Upd]}$$

$$\frac{E \vdash a : A \quad \vdash A \leq B}{E \vdash a : B} \quad \text{[Subs]} \qquad \frac{E \vdash a : \langle R, [R, l : B]\rangle^+}{E \vdash a.l : B} \quad \text{[Meth]}$$

$$\frac{E \vdash a : \langle R, [R_i, l_i : B_i{}^{i\in I}]\rangle^+}{E \vdash a \ \mathbf{as}\ R' : \langle R', [R_i, l_i : B_i{}^{i\in I}]\rangle^+} \quad \text{[Is]} \qquad \frac{E \vdash a : \langle R, [\ ]\rangle^+}{E \vdash a \ \mathbf{is}\ R' : \mathit{bool}} \quad \text{[As]}$$

$$\frac{E \vdash a : \langle R, [\ ]\rangle^+}{E \vdash \mathbf{check}(a : A) : A} \quad \text{[Check]}$$

4.3 The reduction rules

We define the operational semantics of the language as a deterministic relation between terms and values, where values are defined by the following grammar.

$$\begin{array}{llll} \textbf{Values} & v & ::= & x \mid k \mid \langle R, [R_i, l_i = \varsigma(x_i : A)\ b_i{}^{i \in I}] \rangle \\ & & & \mid \text{AsError} \mid \text{CheckError} \end{array}$$

We have included the operator **as** and $\mathbf{check}(a : A)$ which may raise runtime error, and we chose to include these errors in the set of values, for the sake of simplicity. We only give here the three most important rules.

$$\frac{\begin{array}{l} a \ \rightarrow\ \langle R, [R_i, l_i = \varsigma(x_i : A)\ b_i{}^{i \in I}] \rangle \\ \exists h \in I.\ R_h, l_h = R, l \quad b_h[x_h \leftarrow o] \ \rightarrow\ v \end{array}}{o.l \ \rightarrow\ v} \quad [\text{Meth}]$$

$$\frac{a \ \rightarrow\ \langle R', [R_i, l_i = \varsigma(x_i : A')\ b_i{}^{i \in I}] \rangle \qquad \nexists i \in I, j \in J.\ R_i, l_i = R, l_j}{\begin{array}{l} a + [R, l_j = \varsigma(x_j : A)\ b_j{}^{j \in J}] \\ \quad \rightarrow\ \langle R, [R_i, l_i = \varsigma(x_i : A)\ b_i{}^{i \in I}; R, l_j = \varsigma(x_j : A)\ b_j{}^{j \in J}] \rangle \end{array}} \quad [\text{Ext}]$$

$$\frac{a \ \rightarrow\ \langle R, [R_i, l_i = \varsigma(x_i : A')\ b_i{}^{i \in I}] \rangle \quad \exists h \in I.\ R_h, l_h = R, l}{\begin{array}{l} o.l \leftarrow \varsigma(x : A)\ b \\ \quad \rightarrow\ \langle R, [R_i, l_i = \varsigma(x_i : A)\ b_i{}^{i \in I \setminus \{h\}}; R_h, l_h = \varsigma(x : A)\ b] \rangle \end{array}} \quad [\text{Upd}]$$

4.4 The strong typing theorem

In the full paper we first define an evaluation function *eval* which, given a closed term, applies the definition of the reduction relation and either diverges, or returns a value (maybe an error), or returns a failure when no reduction rule can be applied. Then, we prove the following strong-typing theorem.

Theorem 4.1 (Strong Typing) *Let c be a closed term. If $() \vdash c : C$ then* $\text{eval}(c)$ *does not return a failure.*

The key lemmas prove transitivity elimination, good behavior w.r.t. well-typed substitution, subject reduction.

5 The hierarchical calculus

5.1 The calculus

For reasons of space, we only give a general discussion; please see the full paper for details.

In the base calculus method extraction is simply direct field access plus self substitution, as in [AC96]. This is the most elementary solution, but it forces a lot of code replication, and it introduces the "downward closure" constraint in the object type formation rule. We now study a variant where, if no *Stud*, *Name* method is present, the *Pers*, *Name* method is used instead.

We first define a lookup function $[R_i, l_i : B_i{}^{i \in I}]_{R,l}$, which returns the minimum super-role of R associated with l in $[R_i, l_i : B_i{}^{i \in I}]$, as:

$$\begin{array}{lll} [R_i, l_i : B_i{}^{i \in I}]_{R,l} = \langle R_j, l_j, B_j \rangle & \text{if} & R_j = \min\{R_i \mid R_i \geq R, l_i = l\} \\ [R_i, l_i : B_i{}^{i \in I}]_{R,l} \uparrow & \text{if} & \{R_i \mid R_i \geq R, l_i = l\} \text{ is empty} \\ & & \text{or has several minimal elements} \end{array}$$

This lookup function is then used to type method extraction (rule [MethH]) and to define the semantics of the same operation.

Inheritance is very useful, but creates a "diamond closure" problem, which resembles the classical multiple-inheritance problems of object-oriented languages. Consider a lattice *Top*, R, S, *Bot*, where *Top* and *Bot* are the maximum and minimum elements, and consider an object o with type $\langle R, [\mathit{Top}, l : T;\ R, l : A;\ S, l : B] \rangle^+$. Considering that the strict current role of o may be *Bot*, how can we type $o.l$? With our lookup technique, $o.l$ would fail if no method for Bot, l were defined, hence the simplest solution is to put a "diamond closure" condition in the good formation rule, which forces us to have a method for Bot, l in situations like this one (the same technique has been used in the λ-& calculus [CGL95]).

$$\forall i, j \in I.\ (i \neq j \wedge l_i = l_j \wedge \exists R.\ (R \leq R_i \wedge R \leq R_j)) \ \Rightarrow\ [R_i, l_i : B_i{}^{i \in I}]_{R,l_i} \downarrow$$ [1]

However, this solution is not acceptable here, since, in the presence of a common subtype T of students and employees, it would force any object which is both a student and an employee to belong to type T too, which is too restrictive for our purposes. Moreover, this solution breaks a hidden assumption of our calculus, which we call "downward openness" of $\mathcal{R}$. We want every term which is well-typed with a given $\mathcal{R}$ to remain well-typed if a new element is added to $\mathcal{R}$, provided that this new element is not a super-role of any old R in $\mathcal{R}$. This weakening-like property allows this calculus to be easily extended with an operation to define new role-tags at the bottom of the current hierarchy, hence to be the foundation of incremental type-checking techniques. This property is enjoyed by all our rules, but would be broken by this diamond closure condition: the previous type is well formed when *Bot* is not in $\mathcal{R}$, but would become ill-formed after *Bot* is added.

Hence we adopt a different solution. Every object in the hierarchical calculus carries both a current role and a set of admissible roles; the syntax is now $\langle R, \{R_j\}^{j \in J}, [R_i, l_i = \varsigma(x_i : A)\ b_i{}^{i \in I}] \rangle$. An object can only assume one of the admissible roles $\{R_j\}^{j \in J}$. Hence, going back to the previous example, when we

[1] If we drop the $i \neq j$ condition, this can be expressed as: $\forall i.\ \forall R \leq R_i.\ [R_i, l_i : B_i{}^{i \in I}]_{R,l_i} \downarrow$, which shows that diamond closure is stricly related to the downward closure problem.

build an object whose type is $\langle R, \{Top, R, S\}, [Top, l : T;\ R, l : A;\ S, l : B]\rangle$, there is no need to define a method for the Bot, l pair, since the operation o **as** Bot is prevented by this type. If we put Bot into the admissible types, then we also have to define a method for Bot, l; this is enforced by the sixth premise of the [ObjFormH] rule.

$$\frac{\begin{array}{ll}(1) & R \in \{R_k\}^{k \in K} \\ (2) & \{R_i\}^{i \in I} \subseteq \{R_k\}^{k \in K} \\ (3) & \forall i \neq j.\ \langle R_i, l_i\rangle \neq \langle R_j, l_j\rangle \\ (4) & \forall i \in I.\ \vdash B_i \diamondsuit \\ (5) & \forall i, j \in I.\ R_i \leq R_j, l_i = l_j \ \Rightarrow \vdash B_i \leq B_j \\ (6) & \forall k \in K.\ \forall i \in I.\ R_k \leq R_i \ \Rightarrow\ [R_i, l_i : B_i{}^{i \in I}]_{R_k, l_i} \downarrow\end{array}}{\vdash \langle R, \{R_k\}^{k \in K}, [R_i, l_i : B_i{}^{i \in I}]\rangle \diamondsuit} \quad \text{[ObjFormH]}$$

We also present two other crucial rules, which define subtyping and method extraction.

$$\frac{\begin{array}{l}\forall i' \in I'.\ \vdash B'_{i'} \diamondsuit \qquad R' \leq R \qquad \{R_k\}^{k \in K} \subseteq \{R'_k\}^{k \in K'} \\ \forall i \in I.\ \exists i' \in I'.\ R'_{i'} \geq R_i, l_{i'} = l_i, \vdash B'_{i'} \leq B_i\end{array}}{\begin{array}{l}\vdash \langle R', \{R'_k\}^{k \in K'}, [R'_i, l'_i : B'_i{}^{i \in I'}]\rangle^+ \\ \qquad \leq \langle R, \{R_k\}^{k \in K}, [R_i, l_i : B_i{}^{i \in I}]\rangle^+\end{array}} \quad \text{[WeakSubH]}$$

$$\frac{\begin{array}{l}E \vdash a : \langle R, \{R_k\}^{k \in K}, [R_i, l_i : B_i{}^{i \in I}]\rangle^+ \\ {[R_i, l_i : B_i{}^{i \in I}]}_{R,l} = \langle R_h, l_h, B_h\rangle\end{array}}{E \vdash a.l : B_h} \quad \text{[MethH]}$$

5.2 The translation

The hierarchical calculus can be faithfully translated onto the base calculus. We only suggest here the translation; observe how the set of admissible roles plays a key role in this translation.

The translation of $A = \langle R, \{R_k\}^{k \in K}, [R_i, l_i : B_i{}^{i \in I}]\rangle$ is the object type $\langle R, [\text{Complete}([R_i, l_i : B_i{}^{i \in I}], \{R_k\}^{k \in K})]\rangle$, where

$$\text{Complete}([R_i, l_i : B_i{}^{i \in I}], \{R_k\}^{k \in K})$$

contains the signature of every message which an object with type A can understand, and is defined as:

$$\begin{array}{ll}\{R, l : T \ \mid\ \forall k \in K, i \in I & \text{such that } [R_i, l_i : B_i{}^{i \in I}]_{R_k, l_i} \downarrow \\ & \text{and } [R_i, l_i : B_i{}^{i \in I}]_{R_k, l_i} = \langle R, l, T\rangle\}.\end{array}$$

Objects are translated in the same way.

6 Role-tags

We anticipated that role-tags are meant to be a model for Fibonacci generative types. In Fibonacci, a generative type definition (`IsA` T `with` Σ) denotes an object type which is characterized by its supertype T, its signature Σ, and a unique time-stamp generated when the definition is processed. At run-time, the type time-stamp is recorded in each role value, and is used to implement operations such as `Is` T and `As` T (method lookup is implemented in a more efficient way, which makes no use of the time-stamp at method lookup time; see [ABGO93, ADG95]). Because of these time-stamps, types are not always erased at run time; for example, if a polymorphic function or a module is parametrized over an object type, it actually receives the timestamp of that type as a parameter.

A role-tag R represents the hidden time-stamp. We decouple the tag from its signature, to keep the model simpler. We are currently studying extensions to deal with modules and parametric polymorphism. In this context, the explicit presence of the role-tags helps understanding when types can be erased and when they have to be passed around at run time; however, the decoupling of the role-tag from the signature becomes much more problematic.

In the full paper we describe a construction for a set $\mathcal{R}$ where every finite type hierarchy can be "faithfully" embedded, in a "downward open" way.

7 Related work

Objects with roles and an extension operation have been studied in [SS91, PK97, GSR96, BG95, Run92]. Most of these works focus on modeling (in the information system sense), with the notable exception of [BG95], where a formal model is presented. This model follows the database tradition and only describes the data aspects but does not formalize the computation. It also differs from our approach since the role played by an object depends on the static type of the expression which denotes the object itself, i.e. they do not have two different values, in the semantic domain, to denote two different roles of the same object, but the message interpretation mechanism is affected both by the dynamic and by the static type of the object. This approach is interesting, but we find it less expressive, and more complex, than the one described in the present paper.

In [ABGO93, AGO95] the role mechanism of Fibonacci is described, and its semantics is outlined informally. This high-level mechanism is behind the basic calculus that we define here.

Many typed calculi supporting record or object extension have been studied (see, for example, [Rém89, JM88, FM96, Liq97, BBDCL97]). All these papers study how to forbid what we called "incompatible extensions" in the presence of subtyping. Indeed, in the presence of "width subtyping", the static type of an expression contains less fields than those in the denoted record, which makes it impossible to be sure that a field f is not already present, maybe with an incompatible type. The proposed solutions range from the assignment of two types to a record, one exact and the other where fields may be forgotten [FM96], to richer type systems where both the presence and absence of fields may be reported [CM91, Rém89, JM88], to systems where the dependencies among different methods are tracked [Liq97]. Preventing incompatible updates is also a problem for us, but it is not our central concern, hence we will adopt the simple solution proposed in [Ghe90, FM96]. The real focus of our research is a new semantics for object extension and message passing which *allows*, under some conditions, incompatible extensions.

A very interesting work which goes in this direction is presented in the paper [RS98]. In the first order system presented in that paper, an object is made of a method suite where every method is indexed by a number, plus a dictionary which maps names to numbers; methods are accessed by name from the outside and by their internal number from *self*. For example, if a method $m_1 = \varsigma(s)\ s.m_2$ is added to an object whose dictionary maps m_2 to 2, then m_1 is stored as $m_1 = \varsigma(s)\ s.2$. In this way, it is possible to forget the existence of m_2 by width subtyping, and then to add a new field named m_2 with a different type without interfering with the future executions of m_1. Indeed, m_1 will still access the method indexed by 2, while the new m_2 will get a different internal number. A method update operation is also defined such that, when method m_2 mapped to 2 is updated using this operations, then it is really the method with internal index 2 which get changed; in this way, the usual late-binding behavior of self can be obtained.

Their proposal is related to ours. In an imperative version of their system, if a student *johnAsStudent* with an integer code is built, then its code its forgotten by subsumption, and finally it is extended with a code "I1" and the result is bound to *johnAsEmployee*, then two different access paths to the same object are obtained, which are, essentially, two different dictionaries, which are similar to our roles. However, there are some differences. First, roles made through dictionaries have no name, hence there are no ***as*** or ***is*** operations. A subtler and more important difference is better explained by an example. Consider an object o with role P and with a method P, m_1 whose body calls *self*.m_2. In our calculus, if we extend it to two different subroles S_1, S_2 which both implement method m_2, then a call to $(o\ \mathbf{as}\ S_i).m_1$ will, correctly, invoke $S_i.m_2$ for $i = 1, 2$; this is the usual late-binding behavior of self.

In Riecke's and Stone's approach, when we add the version of m_2 for S_1 we use the update operation, to obtain the late-binding behavior of self. Afterwards, when we add the version of m_2 for S_2, we have to choose between extension and method update. If we use extension, we obtain a new dictionary for the object but *self*.m_2, inside m_1, remains bound to the old version of m_2. If we use method override then *self*.m_2 gets bound to the new version of m_2, but there is no way to make it use the old version: with extension we have roles but static binding of self, with method update we have dynamic binding but no roles. This is not, of course, a fault of Riecke and Stone's approach, but just a consequence of the fact that their aim is different from the one of this work.

8 Conclusions

Object extension and roles cannot be avoided in certain applications of object-oriented languages, but these notions lack a solid foundation. We have presented such a foundation and have commented on some of the key issues that arise in our setting: resolution of ambiguous messages, covariance, downward or diamond closure, and extensibility of the set of role tags. Most of these issues are directly related to some of the hardest problems we had to face during the design of the Fibonacci language.

Conclusions

We thanks the anonymous referees for the constructive references. Discussions with Luca Cardelli and John Riecke have been very helpful. This work has been supported in part by grants from the E.U., workgroup PASTEL, and by "Ministero dell'Università e della Ricerca Scientifica e Tecnologica", project INTERDATA.

References

[ABGO93] A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An object data model with roles. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 39–51, Dublin, Ireland, 1993.

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[ADG95] A. Albano, M. Diotallevi, and G. Ghelli. Extensible objects for database evolution: Language features and implementation issues. In *Proceedings of the Fifth Intl. Workshop on Data Base Programming Languages (DBPL), Gubbio, Italy*, 1995.

[AGO95] A. Albano, G. Ghelli, and R. Orsini. Fibonacci: A programming language for object databases. *The VLDB Journal*, 4(3):403–439, 1995.

[BBDCL97] Viviana Bono, Michele Bugliesi, Mariangiola Dezani-Ciancaglini, and Luigi Liquori. Subtyping constraints for incomplete objects. In *Proceedings of TAPSOFT/CAAP 97*, volume 1214 of *Lecture Notes in Computer Science*, pages 465–477. Springer-Verlag, Berlin, 1997.

[BG95] E. Bertino and G. Guerrini. Objects with Multiple Most Specific Classes. In *Proc. Ninth European Conference on Object-Oriented Programming*, LNCS N. 952, pages 102–126, Berlin, 1995. Springer-Verlag.

[CGL95] Giuseppe Castagna, Giorgio Ghelli, and Giuseppe Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, 15 February 1995. a preliminary version appeared in LISP and Functional Programming, July 1992 (pp. 182–192), and as Rapport de Recherche LIENS-92-4, Ecole Normale Supérieure, Paris.

[CM91] Luca Cardelli and John Mitchell. Operations on records. *Mathematical Structures in Computer Science*, 1:3–48, 1991. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994); available as DEC Systems Research Center Research Report #48, August, 1989, and in the proceedings of MFPS '89, Springer LNCS volume 442.

[FM96] Kathleen Fisher and John Mitchell. The development of type systems for object-oriented languages. *Theory and Practice of Object Systems*, 1(3):189–220, 1996.

[Ghe90] Giorgio Ghelli. A class abstraction for a hierarchical type system. In *Proceedings of ICDT 90*, volume 470 of *Lecture Notes in Computer Science*, pages 56–71. Springer-Verlag, Berlin, 1990.

[Ghe91] Giorgio Ghelli. A static type system for message passing. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 129–143, Phoenix, Arizona, October 1991. Distributed as SIGPLAN Notices, Volume 26, Number 11, November 1991.

[GSR96] G. Gottlob, M. Schrefl, and B. Röck. Extending Object-Oriented Systems with Roles. *ACM Transactions on Information Systems*, 14(3):268–296, 1996.

[JM88] Lalita A. Jategaonkar and John C. Mitchell. ML with extended pattern matching and subtypes (preliminary version). In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 198–211, Snowbird, Utah, July 1988.

[Liq97] Luigi Liquori. An extended theory of primitive objects: First order system. In M.Aksit and S. Matsuoka, editors, *Proceedings of ECOOP 97*, volume 1241 of *Lecture Notes in Computer Science*, pages 146–169. Springer-Verlag, Berlin, 1997.

[PK97] M.P. Papazoglou and B.J. Krämer. A Database Model for Object Dynamics. *The VLDB Journal*, (6):73–96, 1997.

[Rém89] Didier Rémy. Typechecking records and variants in a natural extension of ML. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin*, pages

242–249. ACM, January 1989. Also in Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design* (MIT Press, 1994).

[RS98] J.C. Riecke and C.A. Stone. Privacy via subsumption. In *Fifth International Workshop on Foundations of Object-Oriented Programming (FOOL 5)*, January 1998.

[Run92] E.A. Rundensteiner. MultiView: A Methodology for Supporting Multiple Views in Object-Oriented Databases. In *Proc. of the Eighteenth Intl. Conf. on Very Large Data Bases (VLDB), Vancouver, British Columbia, Canada*, pages 187–198, San Mateo, California, 1992. Morgan Kaufmann Publishers.

[SS91] M.H. Scholl and H.-J. Schek. Supporting Views in Object-Oriented Databases. *IEEE Data Engineering Bulletin*, 14(2), 1991.