Foundations for Extensible Objects with Roles

Giorgio Ghelli
Dipartimento di Informatica, Corso Italia, 40, 56125 Pisa, Italy

E-mail: ghelli@di.unipi.it

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 modeled by 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.

Key Words: Object-oriented languages, roles

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. Many foundational studies have been devoted to the problem of defining an object (or record) extension operation which prevents incompatible extensions.

A different approach, studied in the field of database languagues ([19], [8], [28], [5]. . .), is to allow incompatible extensions, by giving a context dependent behavior to the extended object: in our example, in different contexts, *John* will play either the *Student* or the *Employee* role, and will answer 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 ([5]). During this process, we had to make some design choices, and to adopt some typing rules, often without a clear understanding of the different choices, or of their consequences and interplay. Our understanding of the object with roles mechanism was not complete, and this paper tries to fill this gap.

We define here a role calculus, defined as a minimal extension of Abadi-Cardelli ς-calculus [2], which embodies, in an abstract way, the essential features we need in a calculus for extensible objects with roles. The focus of our research is not on the extension operation, but on the good formation properties which allow the different methods introduced by incompatible extensions to coexist, on the semantics of message passing, and on the role of generative types.

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 is formally introduced in Section 4. In Section 5 we prove the main properties of the calculus, subject reduction and strong typing. In Section 6 we show how the calculus can be enriched with an inheritance mechanism, and we describe a translation from the hierarchical to the basic calculus. In Section 7 we discuss an important technical point, the internal structure of the set of role-tags. Section 8 discusses some related works. Section 9 draws some conclusions.

2. THE ς-CALCULUS

Our model is defined as an extension of Abadi-Cardelli ς-calculus [2]. 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 whole object; method update updates the body of a method. The syntax of the Abadi-Cardelli calculus is defined below.

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

The notation $[X_i{}^{i\in 1\ldots n}]$ stands for a sequence $[X_1; \ldots; X_n]$.

The operational semantics is defined by the following evaluation relation.

(Red. Object)

$$\frac{v = [l_i = \varsigma(x_i{:}A)\ b_i\ ^{i\in I}]}{v\ \rightarrow\ v}$$

(Red. Select)

$$\frac{a\ \rightarrow\ [l_i = \varsigma(x_i{:}A)\ b_i\ ^{i\in I}] = o \quad h \in I \quad b_h\{x_h \leftarrow o\}\ \rightarrow\ v}{a.l_h\ \rightarrow\ v}$$

(Red. Update)

$$\frac{a\ \rightarrow\ [l_i = \varsigma(x_i{:}A)\ b_i\ ^{i\in I}] \quad h \in I}{a.l_h \leftarrow \varsigma(x{:}A)\ b\ \rightarrow\ [l_h = \varsigma(x{:}A)\ b, l_i = \varsigma(x_i{:}A)\ b_i\ ^{i\in I\setminus\{h\}}]}$$

The type rules of the calculus are as follows.

(Type Object)

$$\frac{\forall i \in I.\ \vdash B_i\ \diamondsuit}{\vdash [l_i : B_i\ ^{i\in I}]\ \diamondsuit}$$

(Val x)

$$\frac{E, x{:}A, E' \vdash\ \diamondsuit}{E, x{:}A, E' \vdash x : A}$$

(Val Select)

$$\frac{E \vdash a : [l_i : B_i\ ^{i\in I}] \quad h \in I}{E \vdash a.l_h : B_h}$$

(Val Object)

$$\frac{\text{let } A = [l_i : B_i\ ^{i\in I}] \quad \forall i \in I.\ E, x_i{:}A \vdash b_i : B_i}{E \vdash [l_i = \varsigma(x_i{:}A)\ b_i\ ^{i\in I}] : A}$$

(Val Update)

$$\frac{\text{let } A = [l_i : B_i\ ^{i\in I}] \quad E \vdash a : A \quad h \in I \quad E, x{:}A \vdash b : B_h}{E \vdash a.l_h \leftarrow \varsigma(x{:}A)\ b : A}$$

3. AN OVERVIEW OF THE ROLE 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 person and extends it to a student and 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 john to Employee
                     methods IdCode = "I1" end;
```

According to the 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. The *Student* and *Employee* roles both contain an *IdCode* field. Observe that the second extension does not override the first one, hence the relative order of the two operations is irrelevant.

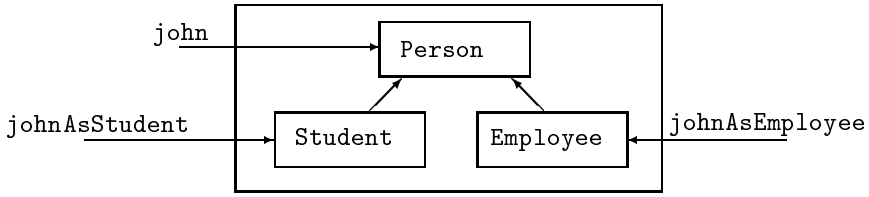

FIG. 1. The internal structure of an object with roles.

When a message is sent to an object, it is actually sent to one of its roles. The corresponding method is then looked for in the receiving role and in its ancestors.

For example, in the previous example `johnAsStudent.Name` invokes the *Name* method from the *Person* role, while `johnAsStudent.IdCode` invokes the *IdCode* method from the *Student* role.

If object extension is never used, then every object is always accessed from its bottom role, and Fibonacci semantics coincides with the standard Smalltalk one. Non standard phenomena only happen after extension, as in the previous example.

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 Fibonacci object (a "role", in Fibonacci jargon), 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 [8]);

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), and 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 hand, 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 in order 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. However, the system we present here models generative types with a non-generative approach, by exploiting the idea of "role-tags"; in Section 7 we give some details on this idea.

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

To model objects with roles we proceed as follows. Since methods are selected on the basis of a message and a role, we extend the Abadi-Cardelli model by indexing methods in an object with a *(role-name,message)* pair, instead of a message only. 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,method suite*⟩ pairs. Hence, an object-with-role playing the role R is now represented as the following pair, where the *current role* R belongs to $\{R_i\}^{\,i\in I}$:

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

The role-tags R_i and R come 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 $\mathcal{R}$ (in a program written in a standard class-based object-oriented language, $\mathcal{R}$ would be the set of the class names ordered by inheritance). For example, the previous example can be modeled by taking

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

where *Ord* is the 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 presented in Section 7.

This syntax allows one to model the *johnAsEmployee* value which is produced by the previous Fibonacci operations as follows.

johnAsEmployee =
$\langle$*Emp*, [(*Pers*, *Name*) $= \varsigma(x:A)$ "John";
(*Stud*, *Name*) $= \varsigma(x:A)$ "John"; (*Stud*, *IdCode*) $= \varsigma(x:A)$ 100;
(*Emp*, *Name*) $= \varsigma(x:A)$ "John"; (*Emp*, *IdCode*) $= \varsigma(x:A)$ "I1"]$\rangle$

Since this basic calculus is modeled over the ς-calculus, it has no inheritance operator, and inheritance can be represented using the same techniques as in [2]. However, the example above shows that here inheritance is more important than in usual object calculi. In fact, 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 example, we have to write down all the three identical methods for *(Pers,Name)*, *(Stud,Name)*, and *(Emp,Name)*, which will be used when the above object will be asked its name through its *Pers*, *Stud*, and *Emp* roles. Later, in Section 6, we will also present a version of the role calculus with inheritance where this redundancy can be avoided, and we will discuss how it can be translated into 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.

For the same reason, as is common in the type-theoretic field, we will define a side-effect-free calculus, where we can study the essential features and avoid some unnecessary

complications. More precisely, though the notion of 'object identity' is not modeled in our calculus, our study will nevertheless face the type-theoretic problems which are posed by identity preserving updates, while avoiding having 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 a functional update of the object should not need to be checked again after the update. Informally, an updated object is referenced both by the instances of *self* in the methods checked before the update and by those in the methods added by the update operation. This form of sharing, though limited, already presents the same type-theoretic challenges that arise in the imperative setting because of the full sharing allowed by the presence of updatable locations. Extending this calculus to an imperative one is relatively straightforward (see [2], Chapters 10-11, but also [7, 22]).

4. THE BASIC CALCULUS

4.1. The syntax

By extending the ς-calculus with the *(role-tag,label)* indexing of methods and by pairing each object with a "current role", we already obtain a kernel role calculus, where most issues can be discussed. We decided, however, to study a calculus which is richer, but more complex, because we want to model all the main Fibonacci role-related operators, hence we extend the calculus with the following additional 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. For the sake of simplicity, we allow at most one new role-tag R to be added by each extension operation, but we have to allow a set of methods $\varsigma(x_i{:}A)\ b_i\ ^{i\in I}$ to be added at once, for reasons which we will discuss later:

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

2. role coercion: the operation o **as** R sets the current role-tag of o to R;

3. role checking: the operation o **is** R tests whether the current role-tag of o is R;

4. dynamic type cast: the operation **check**$(a : A)$ casts a to the object type A, and fails if this is not sound. We model this failure by the propagation of a special value **checkerr**, i.e. **check**$(a : A)$ evaluates to **checkerr** whenever the run-time type of a is not a subtype of A, and f(**checkerr**) evaluates to **checkerr** for every f.

The **check**$(a : A)$ operation is just a simple model of a type-cast (or dynamic typing) facility which is, in practice, very useful in this context. We deal with it for the sake of completeness, but it may be substituted by any other dynamic typing operator, or be dropped altogether, without affecting the rest of the system.

In some approaches, object extension and field update are merged in one operation which either updates the field, when it is already in the object, or adds it, when it is not there. We prefer to keep the two operations separate, both because we want to study their different typing rules, and because we believe that this separation, in a programming language, increases program readability.

We may substitute object construction with empty object construction ($\langle R, [] \rangle$) plus extension. However, we prefer to keep full object construction because we see extension, **as** , **is** , and **check**$(a : A)$, as something which is not in the hard kernel of the system,

hence we prefer to have a system which would remain complete even if we took these operators out.

The syntax of the calculus is thus defined as follows. Hereafter, metavariables R, S_i, R_i, and their primed versions, range over $\mathcal{R}$, while l, m, and l_i range over a denumerable set of labels. The two forms of the object type will be explained in the next subsection. As usual, we consider terms modulo α-equivalence, and the order of fields is irrelevant in objects and in object types.

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

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. R does not depend on i in the object extension operation since an object acquires at most one new role at a time.

Hereafter we will use the following notation:

- if $A = \langle R, [(R_i, l_i) : B_i{}^{i \in I}] \rangle$, then $A^+ = \langle R, [(R_i, l_i) : B_i{}^{i \in I}] \rangle^+$.
- if $A = \langle R, [(R_i, l_i) : B_i{}^{i \in I}] \rangle^+$, then $A^- = \langle R, [(R_i, l_i) : B_i{}^{i \in I}] \rangle$.

4.2. Typing and subtyping

As in [2], the type of an object describes the structure of the object itself, hence its syntax is $\langle R, [(R_i, l_i) : B_i{}^{i \in I}] \rangle$. On these types, 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. Subsumption combined with width subtyping implies that the type of an object o only records a subset of its actual fields, which makes it impossible to statically check some good formation properties of objects built by extending o. 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 [StrictForm] below), and strict types are used to type the extension operation (rule [Ext]). The weak object type $\langle R, [(R_i, l_i) : B_i{}^{i \in I}] \rangle^+$ only lists some messages which are guaranteed to be answered by the object, hence width subtyping applies to weak types, 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 depth subtyping on weak types (rule [WeakDepthSub]) (depth subtyping means that the type of a method in a subtype is generally not equal but just a subtype of the type of the same method in the supertype; depth subtyping is not compatible with method update operations). Strict types can be

promoted to the corresponding 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 [16], and developed independently, for object update, in [14, 15]; it is also strictly connected with the idea of "row variables" [30].

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 define a function to print the name of a person as in the following two lines, written in a role-based toy-language, and then to apply that function to students and employees.

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

However, role promotion creates a soundness problem. It would not be sound to pass an object o whose strict type is $\langle \textit{Stud}, [(\textit{Pers}, \textit{Name}) : \textit{string}]\rangle$ 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 $\langle \textit{Pers}, [(\textit{Pers}, \textit{Name}) : \textit{string}]\rangle$. 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 third premise of rule [StrictForm], and will come (almost) for free in the version with inheritance. The premise can be 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., $(R_h, l_h) = (R_i, l_j)$). We check this condition for every role-tag R_i which appears in some other method, instead of every role-tag in $\mathcal{R}$, thanks to the condition $R \in \{R_i\}^{i\in I}$ which appears in the [StrictForm] and [As] rules.

A problem would also arise if we allowed an object with strict type $\langle \textit{Stud}, [(\textit{Stud}, \textit{Name}) : \textit{int};\ (\textit{Pers}, \textit{Name}) : \textit{string}]\rangle$ to be passed to the same function. In this case, the $(\textit{Stud}, \textit{Name})$ method answers the call `x.name` which has been typed with respect to the $(\textit{Pers}, \textit{Name})$ 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 second premise of rule [StrictForm]. Notice that this covariance is orthogonal to the depth subtyping question, but is strictly related to the same condition we find in the λ-& calculus of overloaded functions with late binding [17, 11, 9].

We are now ready to present the good formation and subtyping rules of our system. In the [Env] rule, $\mathrm{Dom}(E)$ is the set of all variables x such that, for some A, $x{:}A$ appears in E. We do not state an explicit reflexivity rule, since it is implied (i.e., admissible) by the [StrictSub] and [WeakDepthSub] rules.

Environment formation

$$\frac{}{() \vdash \diamondsuit} \quad [\text{EmptyEnv}] \qquad \frac{E \vdash \diamondsuit \quad \vdash A \diamondsuit \quad x \notin \mathrm{Dom}(E)}{E, x{:}A \vdash \diamondsuit} \quad [\text{Env}]$$

Type formation

$$\frac{\begin{array}{l}(1)\ \ \forall i \neq j.\ (R_i, l_i) \neq (R_j, l_j)\\ (2)\ \ \forall i, j \in I.\ R_i \leq R_j, l_i = l_j\ \Rightarrow \vdash B_i \leq B_j\\ (3)\ \ \forall i, j \in I.\ R_i \leq R_j\ \Rightarrow\ \exists h \in I.\ (R_h, l_h) = (R_i, l_j)\\ (4)\ \ R \in \{R_i\}^{\,i \in I}\end{array}}{\vdash \langle R, [(R_i, l_i) : {B_i}^{\,i \in I}]\rangle\ \diamondsuit} \quad \text{[StrictForm]}$$

$$\frac{\vdash \langle R, [(R_i, l_i) : {B_i}^{\,i \in I}]\rangle\ \diamondsuit}{\vdash \langle R, [(R_i, l_i) : {B_i}^{\,i \in I}]\rangle^{+}\ \diamondsuit} \quad \text{[WeakForm]}$$

Subtyping

$$\frac{\vdash \langle R, [(R_i, l_i) : {B_i}^{\,i \in I}]\rangle\ \diamondsuit}{\vdash \langle R, [(R_i, l_i) : {B_i}^{\,i \in I}]\rangle \leq \langle R, [(R_i, l_i) : {B_i}^{\,i \in I}]\rangle} \quad \text{[StrictSub]}$$

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

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

$$\frac{\vdash A \leq A' \quad \vdash A' \leq A''}{\vdash A \leq A''} \quad \text{[Transitivity]}$$

We can now present the typing rules.

Rules [ObjIntro], [Ext], and [Upd], check that the resulting type is well formed, and that every (new) method has the correct type, under suitable assumptions over the type of *self* (x_j or x). In all these rules, methods are type-checked under the assumption that the type of *self* is a weak version A^+ of the object type. We cannot use the stronger assumption that *self* has the strict type A, since otherwise every method should be re-type-checked any time the object is extended and its type grows. Indeed, observe that, in the [Ext] rule, the A^+ type of *self* after extension is different from the type of a before extension, and from the type of *self* used to type-check the methods of a. This coexistence of different self types is a well-known phenomenon, and an essential feature of most calculi which support object extension. The proof of the compatibility between the actual run-time type of an object and the types of its self variables is the kernel of the proof of the strong typing theorem in Section 5.

In the [ObjIntro] rule, each b_j method is checked under the assumption that x_j has type $\langle R_j, [(R_i, l_i) : {B_i}^{\,i \in I}]\rangle^{+}$, rather than $A^+ = \langle R, [(R_i, l_i) : {B_i}^{\,i \in I}]\rangle^{+}$. We fix the role of x_j

to R_j since we know that, when the method b_j is selected, the role of the receiving object is R_j, hence this will be the role of *self* (x_j). In the [Ext] and [Upd] rules we can write the same assumption as $x{:}A^+$ because, in both cases, the role associated with the method b_j (b, in the [Upd] rule) is exactly the role R which appears in A.

The extension operator is allowed to add many fields at a time, while updating can only update one of them. We need this ability of adding many methods at a time, because the type of the resulting object must be well-formed, and the third well-formedness condition of rule [StrictForm] (downward closure) requires that, when one role R is added to an object, *all* messages which have a method for a superrole of R inside the object acquire a method for R too. We may extend the update operation to update many fields too, but we prefer to keep it simpler.

The [Meth] rule only requires the message l to be understood by the current role R of a. If the type of a contains more fields, we use subsumption to promote the type of a to one which only contains the (R, l) method.

The [As] rule requires R' to be a role for which a has at least one method. This side condition is used to model the notion that an object only has some specific roles (for example, one person is a student, while another one is not), and cannot be casted to a role which the object does not possess. It is also useful, as we said before, to make the downward closure condition more tractable (rule [ObjIntro], condition (3)).

[Is] and [Check] only require a to belong to some object type.

Finally, the [Error] rule gives **checkerr** any type. This happens because **checkerr** behaves likes an exception: any operator can be applied to **checkerr**, and the result is always the propagation of the exception, i.e. the value **checkerr** itself. This may also be modeled by assigning a bottom type to **checkerr**, as happens for example in the Galileo language [3], or by designing a full-fledged exception mechanism.

Term formation

$$\frac{E, x{:}A, E' \vdash \diamondsuit}{E, x{:}A, E' \vdash x : A} \quad \text{[Var]}$$

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

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

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

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

$$\frac{E \vdash a : \langle R, [(R_i, l_i) : B_i{}^{i\in I}]\rangle^+ \quad R' \in \{R_i\}^{i\in I}}{E \vdash a \textbf{ as } R' : \langle R', [(R_i, l_i) : B_i{}^{i\in I}]\rangle^+} \quad \text{[As]}$$

$$\frac{E \vdash a : \langle R, [\,]\rangle^+}{E \vdash a \textbf{ is } R' : \mathit{bool}} \quad \text{[Is]} \qquad \frac{\vdash A \diamondsuit \quad E \vdash a : \langle R, [\,]\rangle^+}{E \vdash \textbf{check}(a : A) : A} \quad \text{[Check]}$$

$$\frac{E \vdash \diamondsuit \quad \vdash A \diamondsuit}{E \vdash \textbf{checkerr} : A} \quad \text{[Error]}$$

We give now an example of a typing derivation for the term

$$(\langle R, [(R, l) = \varsigma(x{:}A)\ true]\rangle + [(S, l) = \varsigma(x{:}B)\ 1]).l$$

where we consider a set $\mathcal{R}$ where R and S are not related, we assume the existence of boolean and integer constants with their types, and we use the following abbreviations:

$$A \;=\; \langle R, [(R, l) : \mathit{bool}]\rangle, \quad B \;=\; \langle S, [(R, l) : \mathit{bool};\ (S, l) : \mathit{int}]\rangle$$

We omit some easy proofs of good formation and subtyping.

(1) $\vdash A = \langle R, [(R, l) : \mathit{bool}]\rangle \diamondsuit$

(2) $x : \langle R, [(R, l) : \mathit{bool}]\rangle^+ \vdash true : \mathit{bool}$

(3) $\vdash (\langle R, [(R, l) = \varsigma(x{:}A^+)\ true]\rangle : \langle R, [(R, l) : \mathit{bool}]\rangle$ by 1, 2, [ObjIntro]

(4) $\vdash B = \langle S, [(R, l) : \mathit{bool};\ (S, l) : \mathit{int}]\rangle \diamondsuit$

(5) $x : \langle S, [(R, l) : \mathit{bool};\ (S, l) : \mathit{int}]\rangle^+ \vdash 1 : \mathit{int}$

(6) $\vdash (\langle R, [(R, l) = \varsigma(x{:}A^+)\ true]\rangle + [(S, l) = \varsigma(x{:}B^+)\ 1])$
$: \langle S, [(R, l) : \mathit{bool};\ (S, l) : \mathit{int}]\rangle$ by 3, 4, 5, [Ext]

(7) $\vdash (\langle R, [(R, l) = \varsigma(x{:}A^+)\ true]\rangle + [(S, l) = \varsigma(x{:}B^+)\ 1])$
$: \langle S, [(S, l) : \mathit{int}]\rangle^+$ by 6, [Subs]

(8) $\vdash (\langle R, [(R, l) = \varsigma(x{:}A^+)\ true]\rangle + [(S, l) = \varsigma(x{:}B^+)\ 1]).l$
$: \mathit{int}$ by 7, [Meth]

4.3. The reduction rules

We now define the operational semantics of the language as a deterministic relation between terms and values, where values are defined by the following grammar, where k includes $true$ and $false$.

$$\textbf{Values} \quad v \;::=\; k \;\mid\; \langle R, [(R_i, l_i) = \varsigma(x_i{:}A)\ b_i{}^{i\in I}]\rangle \;\mid\; \textbf{checkerr}$$

The operator **check**$(a : A)$ may raise run-time errors (exceptions) when well-typed terms are evaluated. We model these errors as special values, which are generated by the application of rule [RCheckErr], and are propagated by the rules [RError] and [RMeth] (when $b_h\{x_h \leftarrow o\}$ reduces to **checkerr**). The propagation of this error does not violate subject reduction, or strong typing, because we decided that **checkerr** has any type. We may say that, by giving **checkerr** any type, we decided that it models those errors which we are not able to prevent by static type-checking. You may compare it with the pseudo-value **crash**, which we introduce in the next section, which has no type and models those errors which are *prevented* by static type-checking. In any case, remember that **check**(a:A) and **checkerr** are not essential to our approach, and the rest of the system does not depend on them in any way.

Observe that the [RExt], [RUpd], and [RAs] rules update the type which is stored inside the object. This is a technical trick, needed to make the system enjoy the subject reduction property. Observe that types are stored inside objects only to support dynamic typing (i.e., the **check**(a:A) operator).

$$\frac{}{v \;\to\; v} \quad \text{[RValue]}$$

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

$$\frac{a \;\to\; \langle R', [(R_i, l_i) = \varsigma(x_i{:}A')\; b'_i{}^{\,i\in I}]\rangle \quad \neg\exists i \in I, j \in J.\; (R_i, l_i) = (R, m_j)}{\begin{array}{l} a + [(R, m_j) = \varsigma(x_j{:}A)\; b_j{}^{\,j\in J}] \\ \;\to\; \langle R, [(R_i, l_i) = \varsigma(x_i{:}A)\; b'_i{}^{\,i\in I}; (R, m_j) = \varsigma(x_j{:}A)\; b_j{}^{\,j\in J}]\rangle \end{array}} \quad \text{[RExt]}$$

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

$$\frac{\begin{array}{l} \text{let } A' = \langle R', [(R_i, l_i) : B_i{}^{\,i\in I}]\rangle \\ a \;\to\; \langle R, [(R_i, l_i) = \varsigma(x_i{:}A)\; b_i{}^{\,i\in I}]\rangle \quad R' \in \{R_i\}^{\,i\in I} \end{array}}{a \textbf{ as } R' \;\to\; \langle R', [(R_i, l_i) = \varsigma(x_i{:}A')\; b_i{}^{\,i\in I}]\rangle} \quad \text{[RAs]}$$

$$\frac{a \;\to\; \langle R, [(R_i, l_i) = \varsigma(x_i{:}A)\; b_i{}^{\,i\in I}]\rangle}{a \textbf{ is } R \;\to\; \mathit{true}} \quad \text{[RIsT]}$$

$$\frac{a \;\to\; \langle R, [(R_i, l_i) = \varsigma(x_i{:}A)\; b_i{}^{\,i\in I}]\rangle \quad R' \neq R}{a \textbf{ is } R' \;\to\; \mathit{false}} \quad \text{[RIsF]}$$

$$\frac{a \;\to\; \langle R, [(R_i, l_i) = \varsigma(x_i{:}A)\; b_i{}^{\,i\in I}]\rangle \quad \vdash A^- \leq A'}{\textbf{check}(a : A') \;\to\; \langle R, [(R_i, l_i) = \varsigma(x_i{:}A)\; b_i{}^{\,i\in I}]\rangle} \quad \text{[RCheck]}$$

$$\frac{a \;\to\; \langle R, [(R_i, l_i) = \varsigma(x_i{:}A)\; b_i\,{}^{i\in I}]\rangle \quad \not\forall\, A^- \leq A'}{\textbf{check}(a : A') \;\to\; \textbf{checkerr}} \quad \text{[RCheckErr]}$$

$$\frac{a \;\to\; \textbf{checkerr}}{C[a] \;\to\; \textbf{checkerr}} \quad \text{[RError]}$$

In the propagation rule [RError], $C[a]$ stands for any of the following expressions: $a.l$, $a.l \leftarrow \varsigma(x{:}A)\; b$, $a + [(R, l_i) = \varsigma(x_i{:}A)\; b_i\,{}^{i\in I}]$, a **as** R, a **is** R, and **check**$(a : A)$.

In the [RCheck] and [RCheckErr] rules, we compare the actual run-time type of the object with A'; the decidability of the subtyping problem is proved in the next section (Corollary 5.1). The run-time type of the object is the strict type A^- which corresponds to the weak type A which is stored as the self type of every object method.

5. THE STRONG TYPING THEOREM

5.1. Strong typing and subject reduction

Strong typing is the property which specifies that the evaluation of a well-typed program will not raise unchecked errors. In our context, strong typing can be *informally* expressed as: $\vdash a : C$ implies that either $\exists v.\; a \;\to\; v$ or the evaluation of a does not terminate.

Strong typing is strictly related to *subject reduction*, i.e. to the fact that, if $\vdash a : C$ is well typed and a reduces to v, then v has type C too. As is customary (see [2]), we will give a real proof of the subject reduction property, which is the interesting kernel of the question, while we will be less formal in the standard transformation from subject reduction to strong typing.

5.2. Subject reduction

To prove subject reduction, we first need some lemmas.

LEMMA 5.1.
$\vdash \langle R, [(R_i, l_i) : B_i\,{}^{i\in I}]\rangle \Diamond,\; R' \in \{R_i\}^{\,i\in I} \;\Rightarrow\; \vdash \langle R', [(R_i, l_i) : B_i\,{}^{i\in I}]\rangle \Diamond$
$\vdash \langle R, [(R_i, l_i) : B_i\,{}^{i\in I}]\rangle^+ \Diamond,\; R' \in \{R_i\}^{\,i\in I} \;\Rightarrow\; \vdash \langle R', [(R_i, l_i) : B_i\,{}^{i\in I}]\rangle^+ \Diamond$

Proof. By the shape of rules [StrictForm] and [WeakForm]. ■

LEMMA 5.2 (Subproof).

1.$E, E' \vdash \Diamond \;\Rightarrow\; E \vdash \Diamond$ *and* $E' \vdash \Diamond$*;*
2.$E \vdash \Diamond$ *and* $E' \vdash \Diamond$ *and* $Dom(E) \cap Dom(E') = \emptyset \;\Rightarrow\; E, E' \vdash \Diamond$*;*
3.$\vdash A \leq B \;\Rightarrow\; \vdash A \Diamond$ *and* $\vdash B \Diamond$*;*
4.$\vdash \langle R, [(R_i, l_i) : B_i\,{}^{i\in I}]\rangle \Diamond \;\Rightarrow\; \forall i \in I. \vdash B_i \Diamond$*;*
5.$\vdash \langle R, [(R_i, l_i) : B_i\,{}^{i\in I}]\rangle^+ \Diamond \;\Rightarrow\; \forall i \in I. \vdash B_i \Diamond$*;*
6.$E \vdash a : A \;\Rightarrow\; E \vdash \Diamond$ *and* $\vdash A \Diamond$*;*
7.$E, x{:}A, E' \vdash \Diamond \;\Rightarrow\; E \vdash \Diamond$ *and* $\vdash A \Diamond$.

Proof. (1,2): by induction on the length of E'. (3): by induction on the proof of $\vdash A \leq B$. (4): by the shape of rule [StrictForm], and by (3). (5): by the shape of rule [WeakForm], and by (4). (6) and (7): by simultaneous induction on the proof of $E \vdash a : A$, $E, x{:}A, E' \vdash \diamondsuit$, and by cases on the last applied rule. For (6), you need (3) for rule [Subs], (5) for rule [Meth], Lemma 5.1 for rule [As]; all the other cases are immediate either by induction or because the thesis is one of the premises of the rule. ■

LEMMA 5.3. *If $E, x{:}A, E' \vdash b : B$ and $\vdash A' \leq A$ then (1) $E, x{:}A', E' \vdash \diamondsuit$ and (2) $E, x{:}A', E' \vdash b : B$.*

Proof. (1): use the Subproof Lemma 5.2. (2): Substitute any application of rule [Var] to x with [Var] plus subsumption, and use (1). ■

LEMMA 5.4 (Generation). *Let $E \vdash c : C$. Then:*

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

- $A = \langle R, [(R_i, l_i) : B_i\ ^{i \in I}] \rangle^+$, *for some* $\{B_i\}\ ^{i \in I}$;
- $\forall j \in I.\ E, x_j{:}\langle R_j, [(R_i, l_i) : B_i\ ^{i \in I}] \rangle^+ \vdash b_j : B_j$;
- $\vdash A^- \leq C$.

2.if $c = a + [R, m_j = \varsigma(x_j{:}A)\ b_j\ ^{j \in J}]$ then

- $A = \langle R, [(R_i, l_i) : B_i\ ^{i \in I}; R, m_j : C_j\ ^{j \in J}] \rangle^+$, *for some* $\{R_i, l_i, B_i\}\ ^{i \in I}$, $\{C_j\}\ ^{j \in J}$;
- $E \vdash a : \langle R', [(R_i, l_i) : B_i\ ^{i \in I}] \rangle$, *for some* R';
- $\forall j \in J.\ E, x_j{:}A \vdash b_j : C_j$;
- $\vdash A^- \leq C$.

3.if $c = a.l \leftarrow \varsigma(x{:}A)\ b$ then

- $A = \langle R, [(R_i, l_i) : B_i\ ^{i \in I}] \rangle^+$, *for some* $\{R_i, l_i, B_i\}\ ^{i \in I}$, R;
- $E \vdash a : \langle R, [(R_i, l_i) : B_i\ ^{i \in I}] \rangle$;
- $\exists h \in I.\ (R_h, l_h) = (R, l)$, $E, x{:}A \vdash b : B_h$;
- $\vdash A^- \leq C$.

4.if $c = a.l$ then $E \vdash a : \langle R, [(R, l) : C] \rangle^+$.

5.if $c = a$ **as** *R' then there exist $\{R_i, l_i, B_i\}\ ^{i \in I}$ such that $R' \in \{R_i\ ^{i \in I}\}$, $\vdash \langle R', [(R_i, l_i) : B_i\ ^{i \in I}] \rangle^+ \leq C$, and $E \vdash a : \langle R, [(R_i, l_i) : B_i\ ^{i \in I}] \rangle^+$ for some R.*

6.if $c = a$ **is** *R' then $C =$* bool *and $E \vdash a : \langle R, [\,] \rangle^+$ for some R.*

7.if $c =$ **check**$(a : A)$ *then $\vdash A \leq C$ and $E \vdash a : \langle R, [\,] \rangle^+$ for some R.*

Proof. The only non syntax-directed rule is [Subs]. For any c, there are exactly two type rules which may be applied to c, [Subs] and the rule r_c which corresponds to the outermost operator of c. Hence, any proof of $E \vdash c : C$ terminates with a proof of $E \vdash c : C'$ by rule r_c, followed by a chain of subsumptions, whose subtyping premises can be grouped by transitivity to form a proof of $\vdash C' \leq C$.

Since we know that $E \vdash c : C'$ has been proved by r_c, we know that the premises of the corresponding instantiation of rule r_c hold; this gives us properties (1), (2), (3), (5), (6), (7). In case (4.), the actual premise of r_c is $E \vdash a : \langle R, [(R,l) : C'] \rangle^+$; $E \vdash a : \langle R, [(R,l) : C] \rangle^+$ follows by $\vdash C' \leq C$, [WeakDepthSub], and subsumption. ■

LEMMA 5.5 (Generation2). *Let* $\vdash A \leq B$. *Then:*

1.if B *is a strict object type, then* $A = B$.

2.if B *is a weak object type* $\langle R, [(R_i, l_i) : {B_i}^{i \in I}] \rangle^+$, *and* A *is a strict object type* $\langle R', [(R'_i, l'_i) : {B'_i}^{i \in I'}] \rangle$ *or a weak object type* $\langle R', [(R'_i, l'_i) : {B'_i}^{i \in I'}] \rangle^+$, *then*

(i)$R' \leq R$;

(ii)$\forall i \in I.\ \exists! i' \in I'.\ (R'_{i'}, l'_{i'}) = (R_i, l_i), \vdash B'_{i'} \leq B_i$

(iii)$\forall i \in I.\ \forall i' \in I'.\ (R'_{i'}, l'_{i'}) = (R_i, l_i) \ \Rightarrow \vdash B'_{i'} \leq B_i$

3.if A *is a strict object type and* B *is a weak object type, then* $\vdash A^+ \leq B$.

Proof. (1.) By induction on the size of the proof, and by cases on the last rule applied, which is either [StrictSub] or [Transitivity].

(ii) We first prove $\forall i \in I.\ \exists i' \in I'.\ (R'_{i'}, l'_{i'}) = (R_i, l_i), \vdash B'_{i'} \leq B_i$, by induction on the size of the proof, and by cases on the last rule applied. [WeakDepthSub], [StrictWeakSub]: immediate. [Transitivity]: if the intermediate type is strict, we conclude by induction and by case (1.). If the intermediate type is $\langle R'', [(R''_i, l''_i) : {B''_i}^{i \in I''}] \rangle^+$, then, by induction:

$$\forall i'' \in I''.\ \exists i' \in I'.\ (R'_{i'}, l'_{i'}) = (R''_{i''}, l''_{i''}), \vdash B'_{i'} \leq B''_{i''}$$
$$\forall i \in I.\ \exists i'' \in I''.\ (R''_{i''}, l''_{i''}) = (R_i, l_i), \vdash B''_{i''} \leq B_i$$

The thesis follows by transitivity. The fact that i' is unique derives from the good formation of A and from the first condition of rule [StrictForm]. (i) is proved in the same way. (iii) is a consequence of (ii).

(3.) By induction on the size of the proof, and by cases on the last rule applied. [StrictWeakSub]: immediate by [WeakDepthSub]. [Transitivity]: if the intermediate type is strict, conclude by induction and case (1.). If the intermediate type is a weak C, $\vdash A^+ \leq C$ by induction, and conclude by transitivity. ■

Lemma 5.5 implies that subtyping is decidable. Consider a relation $\vdash_{alg} A \leq B$ which is defined by the [StrictSub], [WeakDepthSub], [StrictForm], [WeakForm] rules, together with the following one, which substitutes [StrictWeakSub] and [Transitivity].

$$\frac{\begin{array}{l} \vdash_{alg} \langle R', [(R'_i, l'_i) : {B'_i}^{i \in I'}] \rangle \ \diamondsuit \\ \vdash_{alg} \langle R, [(R_i, l_i) : {B_i}^{i \in I}] \rangle^+ \ \diamondsuit \\ R' \leq R \\ \forall i \in I.\ \exists i' \in I'.\ (R'_{i'}, l'_{i'}) = (R_i, l_i) \wedge \vdash_{alg} B'_{i'} \leq B_i \end{array}}{\vdash_{alg} \langle R', [(R'_i, l'_i) : {B'_i}^{i \in I'}] \rangle \leq \langle R, [(R_i, l_i) : {B_i}^{i \in I}] \rangle^+} \quad \text{[AlgStrictWeakSub]}$$

This set of rules is syntax-directed. Moreover, if we measure a subtyping or good formation problem by the sum of the sizes of the involved types, these rules always reduce

a problem to a set of strictly smaller problems. Hence, the $\vdash_{alg} A \leq B$ and $\vdash_{alg} A \Diamond$ problems are decidable.

Rule [AlgStrictWeakSub] is admissible in our system, hence $\vdash_{alg} A \leq B \Rightarrow \vdash A \leq B$. Lemma 5.5 (together with Lemma 5.2, when two strict types are compared) implies that $\vdash A \leq B \Rightarrow \vdash_{alg} A \leq B$. Hence, the following corollary holds.

COROLLARY 5.1 (Decidability of Subtyping). *The subtype problem is decidable.*

We may now use Lemma 5.4 to define a set of syntax-directed rules for the type-checking problem too, and prove the following corollary.

COROLLARY 5.2 (Decidability of Type-Checking). *Type-checking is decidable.*

LEMMA 5.6 (Weakening). *Let* $E, E'' \vdash c : C$ *and* $E, E', E'' \vdash \Diamond$, *then* $E, E', E'' \vdash c : C$.

Proof. By induction on the proof of $E, E'' \vdash c : C$. Notice that we reason modulo α renaming, hence we can rename all the bound variables inside c so that they are different from those defined in E' (see [23] for an alternative approach). ■

LEMMA 5.7 (Substitution). *Let* $E, x{:}A, E' \vdash c : C$ *and* $E \vdash a : A$, *then* $E, E' \vdash c\{x \leftarrow a\} : C$.

Proof. By induction on the proof of $E, x{:}A, E' \vdash c : C$ and by cases on the last applied rule. The only interesting case is rule [Var], where we conclude by Lemma 5.6. ■

LEMMA 5.8 (Subject Reduction). *If* $\vdash c : C$ *and* $c \rightarrow v$ *then* $\vdash v : C$.

Proof. By induction on the size of the proof of $c \rightarrow v$, and by cases on the last rule applied. For the sake of brevity, we will often use Lemma 5.2 (Subproof) and the subsumption rule without mentioning them.

• *[RMeth]:* In this case, $c = a.l$ and $a \rightarrow \langle R, [(R_i, l_i) = \varsigma(x_i{:}A)\ b_i\ ^{i \in I}] \rangle = o$ *(a)*, $\exists h \in I.\ (R_h, l_h) = (R, l)$ *(b)*, and $b_h\{x_h \leftarrow o\} \rightarrow v$ *(c)*.

By Lemma 5.4(4.), $\vdash a : \langle R, [(R, l) : C] \rangle^+$.

By *(a)* and induction hypothesis,

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

By Lemma 5.4(1.):

- $A = \langle R, [(R_i, l_i) : B_i\ ^{i \in I}] \rangle^+$, for some $\{B_i\}^{i \in I}$ *(d)*;
- $\forall j \in I.\ x_j{:}\langle R_j, [(R_i, l_i) : B_i\ ^{i \in I}] \rangle^+ \vdash b_j : B_j$ *(e)*;
- $\vdash A^- \leq \langle R, [(R, l) : C] \rangle^+$ *(f)*.

By *(f)*, *(b)*, and Lemma 5.5(2.), $\vdash B_h \leq C$ *(g)*.

By *(d)*, *(e)*, rule [ObjIntro], $\vdash o : A^-$, hence $\vdash o : A$ *(h)*.

By *(e)*, since $R = R_h$, $x_h{:}\langle R, [(R_i, l_i) : B_i{}^{i\in I}]\rangle^+ \vdash b_h : B_h$, hence by *(h)*, and Lemma 5.7, $\vdash b_h\{x_h \leftarrow o\} : B_h$.

By induction hypothesis, $\vdash v : B_h$, hence, by *(g)*, $\vdash v : C$.

- *[RExt]:* In this case, $\vdash c = a + [(R, m_j) = \varsigma(x_j{:}A)\ b_j{}^{j\in J}] : C$ *(a)*,

  $v = \langle R, [(R_i, l_i) = \varsigma(x_i{:}A)\ b'_i{}^{i\in I}; (R, m_j) = \varsigma(x_j{:}A)\ b_j{}^{j\in J}]\rangle$ and

  $a \ \rightarrow\ \langle R', [(R_i, l_i) = \varsigma(x_i{:}A')\ b'_i{}^{i\in I}]\rangle$ *(b)*,

  $\neg\exists i \in I, j \in J.\ (R_i, l_i) = (R, m_j)$ *(c)*.

By *(a)* and Lemma 5.4(2.):

  – $A = \langle R, [(R'_k, l'_k) : B'_k{}^{k\in K}; (R, m_j) : C_j{}^{j\in J}]\rangle^+$,

for some $\{R'_k, l'_k, B'_k\}^{k\in K}$, $\{C_j\}^{j\in J}$ *(d)*;

  – $\vdash a : \langle R'', [(R'_k, l'_k) : B'_k{}^{k\in K}]\rangle$, for some R'' *(e)*;

  – $\forall j \in J.\ x_j{:}A \vdash b_j : C_j$ *(f)*;

  – $\vdash A^- \leq C$ *(g)*.

By *(b)*, *(e)*, and by the inductive hypothesis,

$\vdash \langle R', [(R_i, l_i) = \varsigma(x_i{:}A')\ b'_i{}^{i\in I}]\rangle : \langle R'', [(R'_k, l'_k) : B'_k{}^{k\in K}]\rangle$.

By Lemma 5.4(1.):

  – $A' = \langle R', [(R_i, l_i) : B''_i{}^{i\in I}]\rangle^+$, for some $\{B''_i\}^{i\in I}$ *(h)*;

  – $\forall j \in I.\ x_i{:}\langle R_j, [(R_i, l_i) : B''_i{}^{i\in I}]\rangle^+ \vdash b'_j : B''_j$ *(i)*;

  – $\vdash (A')^- = \langle R', [(R_i, l_i) : B''_i{}^{i\in I}]\rangle \leq \langle R'', [(R'_k, l'_k) : B'_k{}^{k\in K}]\rangle$ *(j)*.

By *(j)* and Lemma 5.5(1.),

$\langle R', [(R_i, l_i) : B''_i{}^{i\in I}]\rangle = \langle R'', [(R'_k, l'_k) : B'_k{}^{k\in K}]\rangle$.

Hence $A = \langle R, [(R_i, l_i) : B''_i{}^{i\in I}; (R, m_j) : C_j{}^{j\in J}]\rangle^+$.

Hereafter, for $j \in I$, let $A_j = \langle R_j, [(R_i, l_i) : B''_i{}^{i\in I}; (R, m_j) : C_j{}^{j\in J}]\rangle^+$. A_j is well formed by Lemma 5.1.

By rule [WeakDepthSub], $\forall j \in I, \vdash A_j \leq \langle R_j, [(R_i, l_i) : B''_i{}^{i\in I}]\rangle^+$.

Hence, by Lemma 5.3 and by *(i)*, $\forall j \in J.\ x_j{:}A_j \vdash b'_j : B''_j$ *(k)*.

By *(k)*, *(f)*, and by rule [ObjIntro],

$\vdash v = \langle R, [(R_i, l_i) = \varsigma(x_i{:}A)\ b'_i{}^{i\in I}; (R, m_j) = \varsigma(x_j{:}A)\ b_j{}^{j\in J}]\rangle : A^-$.

By *(g)* and subsumption, $\vdash v : C$.

- *[RUpd]:* In this case, $\vdash c = a.l \leftarrow \varsigma(x{:}A)\ b : C$ *(a)*,

  $v = \langle R, [(R_i, l_i) = \varsigma(x_i{:}A)\ b_i{}^{i\in I\setminus\{h\}}; (R_h, l_h) = \varsigma(x{:}A)\ b]\rangle$ for some $h \in I$ *(b)*, and

  $a \ \rightarrow\ \langle R, [(R_i, l_i) = \varsigma(x_i{:}A')\ b_i{}^{i\in I}]\rangle$ *(c)*, and $(R_h, l_h) = (R, l)$ *(d)*.

By *(a)* and Lemma 5.4(3.):

  – $A = \langle R', [(R'_k, l'_k) : B'_k{}^{k\in K}\rangle^+$, for some $\{R'_k, l'_k, B'_k\}^{k\in K}$, R' *(e)*;

  – $\vdash a : \langle R', [(R'_k, l'_k) : B'_k{}^{k\in K}]\rangle$ *(f)*;

– $\exists g \in K.\ (R'_g, l'_g) = (R', l)$, $x{:}A \vdash b : B'_g$ *(g)*;

– $\vdash A^- \leq C$ *(h)*.

By *(f)*, *(c)*, and by the induction hypothesis:

$\vdash \langle R, [(R_i, l_i) = \varsigma(x_i{:}A')\ b_i\ ^{i\in I}] : \langle R', [(R'_k, l'_k) : B'_k\ ^{k\in K}]\rangle\rangle$ *(i)*.

By *(i)* and Lemma 5.4(1.):

– $A' = \langle R, [(R_i, l_i) : B_i\ ^{i\in I}]\rangle^+$, for some $\{B_i\}^{\,i\in I}$ *(j)*;

– $\forall j \in I.\ x_j{:}\langle R_j, [(R_i, l_i) : B_i\ ^{i\in I}]\rangle^+ \vdash b_j : B_j$, *(k)*;

– $\vdash (A')^- \leq \langle R', [(R'_k, l'_k) : B'_k\ ^{k\in K}]\rangle = A^-$ *(l)*.

By *(l)* and Lemma 5.5(1.), $(A')^- = A^-$,

i.e.: $\langle R, [(R_i, l_i) : B_i\ ^{i\in I}]\rangle = \langle R', [(R'_k, l'_k) : B'_k\ ^{k\in K}]\rangle$, $R = R'$, hence, by *(g)* and *(d)*, the following equalities hold: $(R'_g, l'_g) = (R', l) = (R, l) = (R_h, l_h)$.

By unicity of (R, l) pairs in well formed types, and from $(A')^- = A^-$, we can conclude that $B'_g = B_h$, hence *(g)* becomes:

$x{:}\langle R_h, [(R_i, l_i) : B_i\ ^{i\in I}]\rangle^+ \vdash b : B_h$ *(m)*.

By *(k)*, *(m)*, and rule [ObjIntro],

$\vdash v = \langle R, [(R_i, l_i) = \varsigma(x_i{:}A)\ b_i\ ^{i\in I\setminus\{h\}};\ (R_h, l_h) = \varsigma(x{:}A)\ b]\rangle : A^-$.

The thesis follows by *(h)*.

- *[RAs]:* In this case, $c = a$ **as** R' *(a)*,

$$
\begin{array}{ll}
a \ \rightarrow\ \langle R, [(R_i, l_i) = \varsigma(x_i{:}A)\ b_i\ ^{i\in I}]\rangle & \textit{(b)},\\
R' \in \{R_i\}^{\,i\in I} & \textit{(c)},\\
v = \langle R', [(R_i, l_i) = \varsigma(x_i{:}A')\ b_i\ ^{i\in I}]\rangle & \textit{(d)},\\
\text{where } A' = \langle R', [(R_i, l_i) : B_i\ ^{i\in I}]\rangle & \textit{(e)};\\
\text{by Lemma 5.1 } \vdash A' \diamondsuit & \textit{(f)}.
\end{array}
$$

By *(a)* and Lemma 5.4(5.),

$\exists \{R'_k, l'_k, B'_k\}^{\,k\in K}.\ \vdash \langle R', [(R'_k, l'_k) : B'_k\ ^{k\in K}]\rangle^+ \leq C$ *(g)*

and $\vdash a : \langle R'', [(R'_k, l'_k) : B'_k\ ^{k\in K}\rangle^+$ for some R'' *(h)*.

By *(b)*, *(h)*, and induction hypothesis:

$\vdash \langle R, [(R_i, l_i) = \varsigma(x_i{:}A)\ b_i\ ^{i\in I}]\rangle : \langle R'', [(R'_k, l'_k) : B'_k\ ^{k\in K}]\rangle^+$ *(i)*.

By Lemma 5.2: $\vdash A \diamondsuit$ *(j)*.

By Lemma 5.4(1.):

– $A = \langle R, [(R_i, l_i) : B_i\ ^{i\in I}]\rangle^+$, for some $\{B_i\}^{\,i\in I}$ *(k)*;

– $\forall j \in I.\ x_j{:}\langle R_j, [(R_i, l_i) : B_i\ ^{i\in I}]\rangle^+ \vdash b_j : B_j$ *(l)*;

– $\vdash A^- \leq \langle R'', [(R'_k, l'_k) : B'_k\ ^{k\in K}]\rangle^+$, hence $\vdash A \leq \langle R'', [(R'_k, l'_k) : B'_k\ ^{k\in K}]\rangle^+$ *(m)*.

By *(e)* ($A' = \langle R', [(R_i, l_i) : B_i \ ^{i \in I}] \rangle^+$), *(f)*, *(l)*, and rule [ObjIntro],

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

i.e. $\vdash v : A'$ *(n)*.

By Lemma 5.5(2.) and rule [WeakDepthSub], *(m)* implies

$\vdash A' \leq \langle R', [(R'_k, l'_k) : B'_k \ ^{k \in K}] \rangle^+$,

hence the thesis follows from *(n)*, *(g)*, and subsumption.

- *[RIsT], [RIsF]:* In this case, C can only be *bool*, which is also the type of $true$ and $false$.
- *[RCheck]:* In this case, $c =$ **check**$(a : A')$,

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

$\vdash A^- \leq A'$ *(a)*,

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

By Lemma 5.4(7.), $\vdash A' \leq C$ *(b)*, and a is well-typed *(c)*.

By *(c)*, $\langle R, [(R_i, l_i) = \varsigma(x_i{:}A)\ b_i \ ^{i \in I}] \rangle$ is well-typed by induction, hence, by Lemma 5.4(1.) and rule [ObjIntro], its type is A^-. The thesis follows by *(a)*, *(b)*, and by subsumption.

- *[RCheckErr], [RError]* Immediate, since **checkerr** belongs to every well-formed type.

■

5.3. Term evaluation

To state the strong typing property, we first have to define an evaluation algorithm *eval*, which receives a term a and applies (backwards) all the rules which match it. We only report here the most significant cases of the algorithm, in an ML-like language; the other cases would not add anything interesting (Figure 2).
For any term a, *eval(a)* is either a value (maybe **checkerr**), or is **crash**, or, if the evaluation of *eval(a)* loops forever, is undefined. By construction, *eval* enjoys the following property.

PROPOSITION 5.1 (Eval).

$$\begin{array}{lcl} a \ \rightarrow \ v & \Rightarrow & \text{eval}(a) = v \\ \text{eval}(a) = v & \Rightarrow & a \ \rightarrow \ v \end{array}$$

Moreover, *eval(a)* captures the distinction between infinite looping and crashing, hence we can now state the strong typing theorem.

THEOREM 5.1 (Strong Typing). *Let c be a closed term. If $\vdash c : C$ then* eval$(c) \neq$ **crash**.

Proof. If *eval*(c) is undefined, the thesis holds. Otherwise, we can reason by induction on the number of recursive calls that are needed to evaluate *eval*(c), and by cases on the shape of c. The crucial case is $a.l$, while the other ones follow easily from the subject reduction property. We only report here the proof for the $a.l$ and extension cases.

```
fun eval(term) =
case term of
v     => v
| a.l => let val v = eval(a)
         in case v of
              ⟨R, [(R_i, l_i) = ς(x_i:A) b_i ^{i∈I}]⟩
                => if ∃h ∈ I. (R_h, l_h) = (R, l)
                     then eval(b_h{x_h ← v}))
                     else crash
              checkerr    => checkerr
              default     => crash
              end case
| a + [R, m_j = ς(x_j:A) b_j ^{j∈J}]
     => let val v = eval(a)
        in case v of
             ⟨R', [(R_i, l_i) = ς(x_i:A') b'_i ^{i∈I}]⟩
               => if ¬∃i ∈ I, j ∈ J. (R_i, l_i) = (R, m_j)
                    then ⟨R, [(R_i, l_i) = ς(x_i:A) b'_i ^{i∈I}; (R, m_j) = ς(x_j:A) b_j ^{j∈J}]⟩
                    else crash
             checkerr    => checkerr
             default     => crash
             end case
| a as R'
     => let val v = eval(a)
        in case v of
             ⟨R, [(R_i, l_i) = ς(x_i:A) b_i ^{i∈I}]⟩
               => if R' ∈ {R_i} ^{i∈I}
                    then ⟨R', [(R_i, l_i) = ς(x_i:A) b_i ^{i∈I}]⟩
                    else crash
             checkerr    => checkerr
             default     => crash
             end case
...
```

FIG. 2. The evaluation procedure.

- $c = a.l$: By Lemma 5.4(4.):

$$\vdash a : \langle R, [(R, l) : C]\rangle^+ \textit{ (a)}$$

By Proposition 5.1 and subject reduction:

$$\vdash v\ (= \textit{eval}(a)) : \langle R, [(R, l) : C]\rangle^+ \textit{ (b)}$$

Since v is well-typed in an empty environment, it is either $v = \textbf{checkerr}$, or $v = \langle R', [(R_i, l_i) = \varsigma(x_i{:}A)\ b_i{}^{i\in I}]\rangle$ *(c)*, for some $\{R_i, l_i, b_i\}^{i\in I}$. The first case is trivial. In the second case, by Lemma 5.4(1.):

$$\begin{array}{ll} A = \langle R', [(R_i, l_i) : B_i{}^{i\in I}]\rangle^+, \text{ for some } \{B_i\}^{i\in I} & \textit{(d)} \\ \forall j \in I.\ x_j{:}\langle R_j, [(R_i, l_i) : B_i{}^{i\in I}]\rangle^+ \vdash b_j : B_j & \textit{(e)} \\ \vdash A^- \leq \langle R, [(R, l) : C]\rangle^+ & \textit{(f)} \end{array}$$

By rule [ObjIntro], $\vdash v : A^-$ *(g)*.

By *(f)* and Lemma 5.5(2.):

$$\begin{array}{ll} R' \leq R & \textit{(h)} \\ \exists h \in I.\ (R_h, l_h) = (R, l) & \textit{(i)} \end{array}$$

By *(i)*, *eval*$(a.l)$ = *eval*$(b_h\{x_h \leftarrow v\})$ *(j)*.

By *(e)* and $R_h = R$,

$$x_h{:}\langle R, [(R_i, l_i) : B_i{}^{i\in I}]\rangle^+ \vdash b_h : B_h \textit{ (k)}$$

By *(h)*, [WeakDepthSub], [StrictWeakSub], and transitivity,

$$\vdash A^- = \langle R', [(R_i, l_i) : B_i{}^{i\in I}]\rangle \leq \langle R, [(R_i, l_i) : B_i{}^{i\in I}]\rangle^+ \textit{(l)}$$

By Lemma 5.3,

$$x_h{:}A^- \vdash b_h : B_h \textit{ (m)}$$

By *(g)*, and Lemma 5.7,

$$\vdash b_h\{x_h \leftarrow v\} : B_h$$

Hence, by induction, *eval*$(b_h\{x_h \leftarrow v\}) \neq$ **crash**. The thesis follows by *(j)*.

- $c = a + [(R, m_j) = \varsigma(x_j{:}A)\ b_j{}^{j\in J}]$: By Lemma 5.4(2.):

$$\begin{array}{ll} A = \langle R, [(R'_k, l'_k) : B'_k{}^{k\in K}; R, m_j : C_j{}^{j\in J}]\rangle^+ & \\ \qquad \text{for some } \{R'_k, l'_k, B'_k\}^{k\in K},\ \{C_j\}^{j\in J} & \textit{(a)} \\ \vdash a : \langle R'', [(R'_k, l'_k) : B'_k{}^{k\in K}]\rangle \ \text{ for some } R'' & \textit{(b)} \end{array}$$

By subject reduction,

$$\vdash v\ (= \mathit{eval}(a)) : \langle R'', [(R'_k, l'_k) : B'_k{}^{\,k\in K}]\rangle \textit{ (c)}$$

Reasoning as above, either $v =$ **checkerr** or

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

In the first case, the result is immediate. In the second case, by the same reasoning as in the corresponding case of the subject reduction proof, we prove that:

$$\langle R'', [(R'_k, l'_k) : B'_k{}^{\,k\in K}]\rangle = \langle R', [(R_i, l_i) : B_i{}^{\,i\in I}]\rangle, \text{ for some } \{B_i\}^{\,i\in I} \textit{ (d)}$$

By *(a)* and *(d)*, $\neg\exists i \in I, j \in J.\ (R_i, l_i) = (R, m_j)$ is a consequence of the good formation of A.

■

6. THE HIERARCHICAL CALCULUS

6.1. The calculus

In the basic calculus method invocation is interpreted as a field access plus self substitution, as in [2]. 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 introduce here a variant where, if no (*Stud*, *Name*) method is present, the (*Pers*, *Name*) method is used instead.

We first define the lookup function

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

which either finds the minimum super-role of R associated with l in $[(R_i, l_i) = \varsigma(x_i{:}A)\, b_i{}^{\,i\in I}]$, or is not defined ($\uparrow$). The function is defined as follows:

$$[(R_i, l_i) = \varsigma(x_i{:}A)\ b_i{}^{\,i\in I}]_{R,l} = \begin{cases} \langle R_j, l_j, b_j\rangle & \text{if } R_j = \min\{R_i \mid R_i \geq R, l_i = l\} \\ \uparrow & \text{if } \{R_i \mid R_i \geq R, l_i = l\} \text{ is empty} \\ & \text{or has several minimal elements} \end{cases}$$

The corresponding lookup function on object types:

$$[(R_i, l_i) : B_i{}^{\,i\in I}]_{R,l}$$

is defined in the same way:

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

These lookup functions are then used to define the semantics of method invocation: when a message l is sent to an object with current role R and method suite $[(R_i, l_i) =$

$\varsigma(x_i{:}A)\ b_i\ ^{i\in I}]$, the method selected is the third component of $[(R_i, l_i) = \varsigma(x_i{:}A)\ b_i\ ^{i\in I}]_{R,l}$ (see Section 6.3).

This form of inheritance is very useful. For example, if you consider the representation of the *johnAsEmployee* object given in Section 3.2, now the *(Stud,Name)* and *(Emp,Name)* methods can be avoided, since the corresponding messages while be answered in the same way by the *(Pers,Name)* method. However, this form of inheritance 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 actual 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 [11]). This condition may be expressed by stating that a type $\langle R, [(R_i, l_i) : B_i\ ^{i\in I}]\rangle$ can be well formed only if, whenever a method for a message l is defined for two different roles $R_i\ R_j$ with a common subrole R, then a method for l is defined for R as well (hereafter, $[(R_i, l_i) = \varsigma(x_i{:}A)\ b_i\ ^{i\in I}]_{R,l}\downarrow$ means that the lookup result is defined):

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

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 as well, 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 that is well-typed with a given $\mathcal{R}$ to remain well-typed if a new element has been 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 type

$$\langle R, [(\mathit{Top}, l) : T;\ (R, l) : A;\ (S, l) : B]\rangle$$

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 R and a set of "roles it belongs to", $\{S_k\}^{k\in K}$; the syntax of an object is now $\langle R, \{S_k\}^{k\in K}, [(R_i, l_i) = \varsigma(x_i{:}A)\ b_i\ ^{i\in I}]\rangle$. An object can only assume one of its $\{S_k\}^{k\in K}$ roles. Hence, going back to the previous example, when we build an object whose type is

$$\langle R, \{\mathit{Top}, R, S\}, [(\mathit{Top}, l) : T;\ (R, l) : A;\ (S, l) : B]\rangle,$$

[1] Due to our definition of the lookup function, this condition is equivalent to the following one: $\forall i \in I.\ \forall R \leq R_i.\ [(R_i, l_i) : B_i\ ^{i\in I}]_{R,l_i}\downarrow$, which shows that diamond closure is strictly related to the downward closure problem.

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 $\{S_k\}^{k\in K}$ roles, then we also have to define a method for *Bot*, l; this is enforced by the fifth premise of the [ObjFormH] rule.

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

The rules of the hierarchical system are reported in the Appendix. We only present here the rules which change; all the other rules are essentially the same as in the basic system. The [WeakDepthSubH] rules can be read as follows. The weak object type A' is a subtype of A if:

- both types are well formed;
- the current role of A' is a subrole of the one of A, (as in the non-hierarchical calculus);
- A' belongs to every role to which A belongs: objects in the subtype may play any role which is played by an object in the supertype;
- for every message (R_i, l_i) which is answered by an object in A, there is a method with index $(R'_{i'}, l'_{i'})$ which can answer the same message, and which returns a value whose type $B'_{i'}$ is compatible with the expected type B_i.

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

$$\frac{\begin{array}{l}\text{let } A = \langle R, \{S_k\}^{k\in K} \cup \{R\}, [(R_i, l_i) : B_i{}^{i\in I}; (R, m_j) : C_j{}^{j\in J}]\rangle\\ E \vdash_h a : \langle R', \{S_k\}^{k\in K}, [(R_i, l_i) : B_i{}^{i\in I}]\rangle \quad \vdash_h A \diamondsuit\\ \forall j \in J.\ E, x_j{:}A^+ \vdash_h b_j : C_j\end{array}}{E \vdash_h a + [(R, m_j) = \varsigma(x_j{:}A^+)\ b_j{}^{j\in J}] : A} \quad \text{[ExtH]}$$

6.2. The translation

Most of the hierarchical calculus can be faithfully translated into the base calculus, by exploiting the set of roles to which an object belongs. However, a problem arises with the extension operation, as we will discuss later.

The translation of an object type contains the signature of every message that the object type understands. It is defined as follows, for weak and strict object types (the construction is similar to the completion construction used in [10] to define a denotational semantics for a version of the λ-& calculus).

$$
\begin{array}{l}
[\![\langle R, \{S_k\}^{k\in K}, [(R_i, l_i) : B_i{}^{i\in I}]\rangle]\!] \\
\quad = \langle R, [\{(R, l) : [\![B]\!] \mid \exists k \in K, i \in I \\
\qquad\qquad\qquad\qquad \text{such that } [(R_i, l_i) : B_i{}^{i\in I}]_{S_k, l_i} \downarrow \\
\qquad\qquad\qquad\qquad \text{and } [(R_i, l_i) : B_i{}^{i\in I}]_{S_k, l_i} = \langle R, l, B\rangle\}]\rangle \\
[\![A^+]\!] = [\![A]\!]^+
\end{array}
$$

The translation of an object is defined in the same way:

$$
\begin{array}{l}
[\![\langle R, \{S_k\}^{k\in K}, [(R_i, l_i) = \varsigma(x_i{:}A)\ b_i{}^{i\in I}]\rangle]\!] \\
\quad = \langle R, [\{(R, l) = \varsigma(x{:}[\![A]\!])\ [\![b]\!] \\
\qquad\qquad \mid \exists k \in K, i \in I \\
\qquad\qquad \text{such that } [(R_i, l_i) = \varsigma(x_i{:}A)\ b_i{}^{i\in I}]_{S_k, l_i} \downarrow \\
\qquad\qquad \text{and } [(R_i, l_i) = \varsigma(x_i{:}A)\ b_i{}^{i\in I}]_{S_k, l_i} = \langle R, l, \varsigma(x{:}A)\ b\rangle\}]\rangle
\end{array}
$$

The rest of the language is translated in the obvious way:

$$
\begin{array}{lcl}
[\![x]\!] & = & x \\
[\![k]\!] & = & k \\
[\![o.l]\!] & = & [\![o]\!].l \\
[\![o.l \leftarrow \varsigma(x{:}A)\ b]\!] & = & [\![o]\!].l \leftarrow \varsigma(x{:}[\![A]\!])\ [\![b]\!] \\
[\![o \textbf{ as } R]\!] & = & [\![o]\!] \textbf{ as } R \\
[\![o \textbf{ is } R]\!] & = & [\![o]\!] \textbf{ is } R \\
[\![\textbf{check}(a : A)]\!] & = & \textbf{check}([\![a]\!] : [\![A]\!]) \\
[\![\textbf{checkerr}]\!] & = & \textbf{checkerr} \\
[\![o + [(R, l_i) = \varsigma(x_i{:}A)\ b_i{}^{i\in I}]]\!] & = & [\![o]\!] + [(R, l_i) = \varsigma(x_i{:}[\![A]\!])\ [\![b_i]\!]^{\,i\in I}]
\end{array}
$$

The idea behind the translation is that the set of methods in an object, or in an object type, is *completed* with respect to the set $\{S_k\}^{k\in K}$, where the completion adds a method, or a method type, for each pair R, l such that the object, while playing the role R, would be able to answer the message l by inheritance.

The translation we have presented, if extended in the obvious way to environments, and if the object extension operation is not used, satisfies the following property, where $\vdash_h$ means that the corresponding judgment has been proved in the hierarchical system:

$$
\begin{array}{lcl}
E \vdash_h \ \diamondsuit & \Rightarrow & [\![E]\!] \vdash \ \diamondsuit \\
\vdash_h A\ \diamondsuit & \Rightarrow & \vdash [\![A]\!]\ \diamondsuit \\
\vdash_h A \leq B & \Rightarrow & \vdash [\![A]\!] \leq [\![B]\!] \\
E \vdash_h a : A & \Rightarrow & [\![E]\!] \vdash [\![a]\!] : [\![A]\!]
\end{array}
$$

Extension does not enjoy this property since, when an object is extended to a new role R, the hierarchical rule does not force all the new methods for R to be specified, since they can be inherited. Hence, the translation of a term $o + [(R, l_i) = \varsigma(x_i{:}A)\ b_i{}^{i\in I}]$ may not contain some methods whose explicit specification is required, in the basic system, because of the lack of inheritance (formally, these methods are needed in the basic system because of the $\vdash A\ \diamondsuit$ premise of rule [Ext], and of the downward closure condition (4) of rule [StrictForm]). When an object is created we overcome the same problem by copying the body of the inherited methods during the translation, but when an object o is extended the bodies of the inherited methods are not necessarily part of the term o (consider, for example, the translation of "$x + \ldots$"). Hence, the translation above always produces well

typed terms only if the source term respects the following stronger rule for extension, where we have added a downward closure condition as the last premise.

$$\frac{\begin{array}{l}\text{let } A = \langle R, \{S_k\}^{k\in K} \cup \{R\}, [(R_i, l_i) : B_i{}^{i\in I}; (R, m_j) : C_j{}^{j\in J}]\rangle \\ E \vdash_h a : \langle R', \{S_k\}^{k\in K}, [(R_i, l_i) : B_i{}^{i\in I}]\rangle \quad \vdash_h A \diamondsuit \\ \forall j \in J.\ E, x_j{:}A^+ \vdash_h b_j : C_j \\ \forall i \in I.\ R \leq R_i \ \Rightarrow\ \exists j \in J.\ l_i = m_j\end{array}}{E \vdash_h a + [(R, m_j) = \varsigma(x_j{:}A^+)\ b_j{}^{j\in J}] : A} \quad \text{[ModExtH]}$$

This rule is equivalent to the standard one whenever $\{S_k\}^{k\in K}$ contains two distinct immediate superroles of R, since in this case all methods for R have to be explicitly specified in both versions of the calculus (because of the diamond closure condition (5) in rule [ObjFormH]). Hence, only "single inheritance extensions" create translation problems, while "multiple inheritance extensions" do not.

To sum up, we claim that the hierarchical calculus with the modified extension rule can be faithfully translated into the basic calculus, in a way which preserves typing. The restriction on the extension rule is not pleasant but is not a major drawback, since the main issues we are trying to face are the coexistence of different methods for the same message, the good formation conditions, the semantics of message passing, and the modeling of generative types through role-tags, while we are less interested here in the details of the object extension operation.

6.3. Operational semantics

The operational semantics for the hierarchical calculus is defined as for the basic system. The main differences are the new shape of object values, which now contain the set $\{S_k\}^{k\in K}$ of allowed roles, and the new form of the crucial [RMeth] rule, which specifies how methods are searched for inside objects. We only report here this last rule.

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

Although we believe that reduction in the hierarchical and in the basic system correspond, we leave the following property as an open issue.

Conjecture 1.

$$\forall a, v, C.\ \vdash_h a : C \ \Rightarrow\ (a \ \rightarrow_h \ v \ \Leftrightarrow\ [\![a]\!] \ \rightarrow\ [\![v]\!])$$

■

7. ROLE-TAGS

7.1. Role-tags and generative types

We mentioned above 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.

For example, the Fibonacci definitions of Section 3.1, are compiled into something which may be represented as follows, where 101, 102, and 103 are three time-stamps, and the LetSub declaration define the order relation on the time-stamps set.

```
Let Person   = <101, [(101,Name): String]>;
LetSub 102 LessThen 101;
Let Student  = <102, [(101,Name): String;
                      (102,IdCode): Int]>;
LetSub 103 LessThen 101;
Let Employee  = <103, [(101,Name): String;
                       (103,IdCode): String]>;

let john           = <101, {101}, [(101,Name)="John"]>;
let johnAsStudent  = john + <102, [(102,IdCode)=100]>;
let johnAsEmployee = john + <103, [(103,IdCode)="I1"]>;
```

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 [1, 4]). 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 following features of the hidden time-stamp:

- two types are the same only if they have both the same signature and the same time-stamp;
- time-stamps are the only components of a type which are also needed at run-time, to implement the `Is` T and `As` T operations; for this reason, the time-stamp belongs both to the type and to the value level.

A Fibonacci time-stamp is always associated with a specific signature. In this study, 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 in 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.

7.2. Role-tags and incremental compilation

A Fibonacci program can be translated into our model through a two phase process. In the first phase, we collect the set $\mathcal{R}$ of all the object types which are defined in the program, ordered by their subtyping relation. Once $\mathcal{R}$ is known, we can translate the program into our role calculus. This is a "whole program" approach: the program is not type-checked incrementally, but type-checking starts only after all the program is known.

However, we can devise a different, incremental, approach. For the sake of simplicity, we assume that all object types in the source program have different names. Then, we assume that $\mathcal{R}$ is the language generated by the following grammar:

$$L \ ::= \ ().\textit{Identifier} \ \mid \ (L_1, \ldots, L_n).\textit{Identifier}$$

The order relation over $\mathcal{R}$ is the reflexive and transitive closure of the relation defined as:

$$(L_1, \ldots, L_n).\mathrm{X} \leq L_i \qquad (i \in \{1, \ldots, n\})$$

This $\mathcal{R}$ allows a type *T* with no supertype to be translated as *().T*, while a type T with n immediate supertypes T_1,. . . ,T_n, is translated as *(T_1^*,. . . ,T_n^*).T*, where T_i^* is the translation of T_i.

For example, if we consider the diamond *Person*, *Student*, *Employee*, *WorkingStudent*, with the order generated by $S \leq P$, $E \leq P$, $WS \leq S$, $WS \leq E$, the four role-tags would be embedded into $\mathcal{R}$ as follows:

$$\begin{array}{lcl} [\![P]\!] & = & ().P \\ [\![S]\!] & = & (().P).S \\ [\![E]\!] & = & (().P).E \\ [\![WS]\!] & = & ((().P).S, (().P).S).\,WS \end{array}$$

With this approach, there is no need to divide type checking into an $\mathcal{R}$-definition phase followed by the actual translation and type-checking. Hence, this interpretation technique shows that the role calculus can be used to understand incremental type-checking.

8. RELATED WORK

Objects with roles and an extension operation have been studied in [29, 24, 19, 8, 28]. Most of these works focus on studying the best way of representing some aspects of a piece of real world, rather than on formal foundations, with the notable exception of [8], where a formal model is presented. The latter 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 [1, 5] the role mechanism of Fibonacci is described, and its semantics is outlined informally. This high-level mechanism underpins the basic calculus that we define here.

Many typed calculi supporting record or object extension have been studied (see, for example, [25, 20, 15, 21, 6]). All these papers study how to *prevent* what we called "incompatible extensions" in the presence of subtyping. Indeed, in the presence of "width subtyping", the static type of an expression contains fewer 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 of which is exact and the other where fields may be forgotten [15], to richer type systems where both the presence and absence of fields may be reported [12, 25, 13, 20, 26, 18], and to systems where the dependencies among different methods are tracked [21]. Preventing incompatible updates is also a problem for us, but it is not our central concern, hence we adopted the simple solution proposed in [16, 15].

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 [27]. 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 $m_1 = \varsigma(s)\ s.2$. It is thus 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 operation, then it is really the method with an internal index 2 which gets changed; in this way, the usual late-binding behavior of *self* can be obtained.

Their proposal is related to ours. In their system, if a student *johnAsStudent* with an integer code is built, its code is later forgotten by subsumption, and finally the student 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 but more important difference exists, which is better explained by an example. Consider an object o with role P and with a method (P, m) whose body calls *self*.m'. In our calculus, if we extend it to two different subroles S_1, S_2 which both implement method m', then a call to $(o$ **as** $S_i).m$ will correctly invoke $(o$ **as** $S_i).m'$ for $i = 1, 2$; this is the usual late-binding behavior of *self*.

In Riecke and Stone's approach, when we add the version of m' for S_1 we can use the update operation to obtain the late-binding behavior of *self*. Afterwards, when we add the version of m' 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', inside m, remains bound to the old version of m'. If we use method update then *self*.m' gets bound to the new version of m', but there is no way to make the object 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 ours.

9. 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.

Although this research is still going on, we have already learned something about the Fibonacci language. First of all, we found a strict correspondence between the pieces of information that we decided to memorize inside the objects, such as the current role and the set of "allowed role-tags" [4], and those which we need in the minimal model that we developed here, which was not unexpected but was still a confirmation that our previous choices were reasonable. Our basic aim, however, was to understand why we were not able to avoid generative object types during the design of the Fibonacci language,

and whether we can remove them from the language, or we can make them interact with modules, subtyping, and explicit polymorphism in a smooth way. We still need to extend this basic model with type variables to be able to answer these questions, but the role-tags model already helped us during the design of the implementation of type application as time-stamp passing.

Another interesting issue is the formalization of an imperative version of this calculus, to check whether the task is really so straightforward as we imagine.

APPENDIX A

Rules of the hierarchical system

Type formation

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

$$\frac{\vdash_h \langle R, \{S_k\}^{k\in K}, [(R_i, l_i) : B_i{}^{i\in I}]\rangle \diamondsuit}{\vdash_h \langle R, \{S_k\}^{k\in K}, [(R_i, l_i) : B_i{}^{i\in I}]\rangle^+ \diamondsuit} \quad \text{[WeakFormH]}$$

Subtyping

$$\frac{\vdash_h \langle R, \{S_k\}^{k\in K}, [(R_i, l_i) : B_i{}^{i\in I}]\rangle \diamondsuit}{\vdash_h \langle R, \{S_k\}^{k\in K}, [(R_i, l_i) : B_i{}^{i\in I}]\rangle \leq \langle R, \{S_k\}^{k\in K}, [(R_i, l_i) : B_i{}^{i\in I}]\rangle} \quad \text{[StrictSubH]}$$

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

$$\frac{\vdash_h \langle R, \{S_k\}^{k\in K}, [(R_i, l_i) : B_i{}^{i\in I}]\rangle \diamondsuit}{\begin{array}{l}\vdash_h \langle R, \{S_k\}^{k\in K}, [(R_i, l_i) : B_i{}^{i\in I}]\rangle\\ \qquad \leq \langle R, \{S_k\}^{k\in K}, [(R_i, l_i) : B_i{}^{i\in I}]\rangle^+\end{array}} \quad \text{[StrictWeakSubH]}$$

Term formation

$$\frac{\begin{array}{l}\text{let } A = \langle R, \{S_k\}^{k\in K}, [(R_i, l_i) : B_i{}^{i\in I}]\rangle\\ \vdash_h A \diamondsuit\\ \forall j \in I.\ E, x_j{:}\langle R_j, \{S_k\}^{k\in K}, [(R_i, l_i) : B_i{}^{i\in I}]\rangle^+ \vdash_h b_j : B_j\end{array}}{E \vdash_h \langle R, \{S_k\}^{k\in K}, [(R_i, l_i) = \varsigma(x_i{:}A^+)\ b_i{}^{i\in I}]\rangle : A} \quad \text{[ObjIntroH]}$$

$$\frac{\begin{array}{l}\text{let } A = \langle R, \{S_k\}^{k\in K} \cup \{R\}, [(R_i, l_i) : B_i{}^{i\in I}; (R, m_j) : C_j{}^{j\in J}]\rangle \\ E \vdash_h a : \langle R', \{S_k\}^{k\in K}, [(R_i, l_i) : B_i{}^{i\in I}]\rangle \quad \vdash_h A \diamondsuit \\ \forall j \in J.\ E, x_j{:}A^+ \vdash_h b_j : C_j\end{array}}{E \vdash_h a + [(R, m_j) = \varsigma(x_j{:}A^+)\ b_j{}^{j\in J}] : A} \quad \text{[ExtH]}$$

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

$$\frac{E \vdash_h a : A \quad \vdash_h A \leq B}{E \vdash_h a : B} \quad \text{[Subs]}$$

$$\frac{E \vdash_h a : \langle R, \{S_k\}^{k\in K}, [(R, l) : B]\rangle^+}{E \vdash_h a.l : B} \quad \text{[MethH]}$$

$$\frac{E \vdash_h a : \langle R, \{S_k\}^{k\in K}, [(R_i, l_i) : B_i{}^{i\in I}]\rangle^+ \quad R' \in \{S_k\}^{k\in K}}{E \vdash_h a \textbf{ as } R' : \langle R', \{S_k\}^{k\in K}, [(R_i, l_i) : B_i{}^{i\in I}]\rangle^+} \quad \text{[AsH]}$$

$$\frac{E \vdash_h a : \langle R, \{\}, [\,]\rangle^+}{E \vdash_h a \textbf{ is } R' : \mathit{bool}} \quad \text{[IsH]} \qquad \frac{E \vdash_h a : \langle R, \{\}, [\,]\rangle^+}{E \vdash_h \textbf{check}(a : A) : A} \quad \text{[CheckH]}$$

ACKNOWLEDGMENT

A preliminary version of this paper was prepared together with Debora Palmerini. Discussions with Luca Cardelli and John Riecke have been very helpful. We also thanks the anonymous referees for many useful suggestions. This work has been supported in part by grants from the E.U., workgroups PASTEL and APPSEM, and by "Ministero dell'Università e della Ricerca Scientifica e Tecnologica", projects INTERDATA and DATA-X.

REFERENCES

1. 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.
2. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
3. A. Albano, L. Cardelli, and R. Orsini. Galileo: A strongly typed, interactive conceptual language. *ACM Transactions on Database Systems*, 10(2):230–260, 1985. Also in S.B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*, Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.
4. A. Albano, M. Diotallevi, and G. Ghelli. Extensible objects for database evolution: Language features and implementation issues. In *Proc. of the 5th Intl. Workshop on Data Base Programming Languages (DBPL), Gubbio, Italy*, 1995.
5. A. Albano, G. Ghelli, and R. Orsini. Fibonacci: A programming language for object databases. *The VLDB Journal*, 4(3):403–439, 1995.
6. V. Bono, M. Bugliesi, M. Dezani-Ciancaglini, and L. Liquori. Subtyping constraints for incomplete objects. In *Proceedings of TAPSOFT/CAAP 97*, volume 1214 of *LNCS*, pages 465–477, Berlin, 1997. Springer-Verlag.
7. V. Bono and K. Fisher. An imperative, first-order calculus with object extension. In *Proc. of 12th European Conference on Object-Oriented Programming (ECOOP), Brussels, Belgium*, volume 1445 of *LNCS*, pages 462–497, Berlin, 1998. Springer-Verlag.

8. E. Bertino and G. Guerrini. Objects with multiple most specific classes. In *Proc. of the 9th European Conference on Object-Oriented Programming (ECOOP), Åarhus, Denmark*, volume 952 of *LNCS*, pages 102–126, Berlin, 1995. Springer-Verlag.

9. G. Castagna. Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 17(3):431–447, 1995.

10. G. Castagna, G. Ghelli, and G. Longo. A semantics for $\lambda\&$-early: A calculus with overloading and early binding. In H. Barendregt, editor, *Proc. of the first International Conference on Typed Lambda Calculi and Applications (TLCA), Utrecht, Olanda*, LNCS, Berlin, March 1993. Springer-Verlag.

11. G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, 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.

12. L. Cardelli and J. Mitchell. Operations on records. *Mathematical Structures in Computer Science (MFCS)*, 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 proc. of MFPS '89, Springer LNCS volume 442.

13. K. Fisher, F. Honsell, and J. C. Mitchell. A lambda calculus of objects and method specialization. *Nordic J. Computing (*formerly *BIT)*, 1:3–37, 1994. Preliminary version appeared in *Proc. of IEEE Symp. on Logic in Computer Science*, 1993, 26–38.

14. K. Fisher and J. Mitchell. A delegation-based object calculus with subtyping. In *Proc. of Intl. Symposium on Fundamentals of Computation Theory (FCT), Dresden, Germany*, volume 965 of *LNCS*, pages 42–61, Berlin, 1995. Springer-Verlag.

15. K. Fisher and J. Mitchell. The development of type systems for object-oriented languages. *Theory and Practice of Object Systems (TAPOS)*, 1(3):189–220, 1995.

16. G. Ghelli. A class abstraction for a hierarchical type system. In *Proc. of Intl. Conference of Database Theory (ICDT)*, volume 470 of *LNCS*, pages 56–71, Berlin, 1990. Springer-Verlag.

17. G. Ghelli. A static type system for message passing. In *Proc. of Intl. Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Phoenix, Arizona*, volume 26(11) of *ACM SIGPLAN Notices*, pages 129–143. ACM, 1991.

18. P. Di Gianantonio, F. Honsell, and L. Liquori. A lambda calculus of objects with self-inflicted extension. In *Proc. of Intl. Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 33(10) of *ACM SIGPLAN Notices*, pages 166–178. ACM, 1998.

19. G. Gottlob, M. Schrefl, and B. Rock. Extending object-oriented systems with roles. *ACM Transactions on Information Systems*, 14(3):268–296, 1996.

20. L.A. Jategaonkar and J.C. Mitchell. ML with extended pattern matching and subtypes (preliminary version). In *Proc. of ACM Conference on Lisp and Functional Programming (LFP)*, pages 198–211, Snowbird, Utah, July 1988.

21. L. Liquori. An extended theory of primitive objects: First order system. In *Proceedings of ECOOP 97*, volume 1241 of *LNCS*, pages 146–169, Berlin, 1997. Springer-Verlag.

22. F. Lang, P. Lescanne, and L. Liquori. A framework for defining object-calculi. In *Proc. of World Congress on Formal Methods (FM) (Volume II), Toulouse, France*, volume 1709 of *LNCS*, pages 963–982, Berlin, 1999. Springer-Verlag.

23. J. McKinna and R. Pollack. Pure type systems formalized. In *Proc. of the International Conference on Typed Lambda Calculi and Applications (TLCA), Utrecht, The Netherlands*, number 664 in LNCS, pages 289–305, Berlin, 1993. Springer-Verlag.

24. M.P. Papazoglou and B.J. Krämer. A database model for object dynamics. *The VLDB Journal*, 6(2):73–96, 1997.

25. D. Rémy. Typechecking records and variants in a natural extension of ML. In *Proc. of the 17th ACM Symposium on Principles of Programming Languages (POPL), 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).

26. D. Rémy. From classes to objects via subtyping. In *Proc. of Programming Languages and Systems, 7th European Symposium on Programming (ESOP), Lisbon, Portugal*, volume 1381 of *LNCS*, pages 200–220, Berlin, 1998. Springer-Verlag.

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

28. 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.

29. M.H. Scholl and H.-J. Schek. Supporting Views in Object-Oriented Databases. *IEEE Data Engineering Bulletin*, 14(2):43–47, 1991.

30. M. Wand. Complete type inference for simple objects. In *Proc. of the IEEE Symposium on Logic in Computer Science (LICS), Ithaca, New York*, pages 37–44, 1987.