Combining subsumption and binary methods: An object calculus with views

Jérôme Vouillon
Department of Computer and Information Science*
University of Pennsylvania
vouillon@saul.cis.upenn.edu

ABSTRACT

We present an object-oriented calculus which allows arbitrary hiding of methods in prototypes, even in the presence of binary methods and friend functions. This combination of features permits complete control of the interface a class exposes to the remainder of a program (which is of key importance for program readability, security and ease of maintenance), while still allowing complex interactions with other classes belonging to the same module or software component.

This result is made possible by the use of *views*. A view is a name that specifies an interface to an object. A set of views is attached to each object and a method can be invoked either directly or via a view of the object.

1. INTRODUCTION

Information hiding is of key importance for making programs more readable, secure and maintainable. In particular, for abstraction purposes, one would expect to be able to specify freely what methods of a class are exported from a package (or a module) to the remainder of a program. Furthermore, abstraction should not be impeded by the need for complex interaction between objects. For instance, if objects from two different classes are to interact with one other, it should still be possible to hide the methods involved in the interaction. This means that the objects must be able to communicate via methods not present in the interfaces exported by the classes to the rest of the program. (A class or a function having such a privileged access to another class is commonly named a *friend*).

A structural subtyping setting, where methods are accessed by name and method names are not lexically scoped, does not provide such freedom [10]. Indeed, if a method is necessary for the interaction between two classes, it must remain in the interface of the class it belongs to (even though its access may be prevented using existential types [2, 15]).

[1]This work was supported in part by the University of Pennsylvania's Institute for Research in Cognitive Science

On the other hand, this problem does not exist in languages such as C++ or Java, where an explicitly declared inheritance hierarchy defines the subtyping relation between objects. The reason is that in this setting a method is selected according to both its name and the static type of the object. Then, a new method in a class will not conflict with a method of the same name in a superclass. Furthermore, it is also possible to expose to the remainder of the program the fact that an object belongs to a subclass of a given class without having to export any of the methods of this class. Then, a friend function is simply a function which has access to some non-exported methods of a class. Type safety is achieved by ensuring that the function is only applied to objects of this class (or one of its subclasses).

However, such languages usually do not support some advanced features often found in languages using structural subtyping, such as selftype and binary methods [3]. We present in this paper a calculus that aims at combining the best of both worlds: it provides these two features while still allowing arbitrary method hiding. Rather than having classes, it uses more atomic constructions: it is prototype-based, and the notion that an object belongs to a given class is expressed by the fact that it possesses a given *view*. A view is simply a name (corresponding to a class name) giving access to an alternate interface to an object.

Several calculi dealing with similar issues have been proposed previously. Riecke and Stone [18] presented a calculus allowing arbitrary hiding of methods. This calculus only uses structural subtyping and does not have binary methods. More recently, Fisher and Reppy [10] proposed a calculus with support for inheritance-based subtyping. However, this calculus is first-order (that is, it does not have the notion of selftype).

For the sake of clarity, we first present in Section 2 a simplified version of the calculus, introducing most features of the full calculus. Then, in Section 3, we present the full calculus and demonstrate how it can properly type binary methods and friend functions. Type checking and type inference are discussed in section 3.4. Section 4 exposes some limitations of the calculus. Finally, related works are presented in section 5.

Notations

We write $f; (i = e)$ to denote the partial function that behaves exactly as f except for mapping i to e. This notation is not standard, but is very convenient, in particular for writing method tables. We write $f \subseteq g$ to denote that the function g extends the function f, that is, the graph of g is a superset of the graph of f.

2. BASE CALCULUS

2.1 Examples

We introduce the calculus through examples. Let us consider the following *object*[1]:

$$\begin{array}{ll} x_1 ::= \zeta(\mathtt{x} : \alpha)[L_1 | \mathcal{L}_1]^{\varphi_1} & \\ \quad \text{where} & \varphi_1 ::= (\mathtt{ten} \mapsto \mathtt{one}; \mathtt{clone} \mapsto \mathtt{two}) \\ & L_1 ::= (\mathtt{one} = 10; \mathtt{two} = \mathtt{x}) \\ & \mathcal{L}_1 ::= (\mathtt{one} : \mathtt{int}; \mathtt{two} : \alpha) \end{array}$$

It contains two methods. The *external names* of these methods (that is, the name of these methods viewed from outside the object) are `ten` and `clone`. A *dictionary* φ_1 (a finite partial function from method names to method names) is used to map these external names to some *internal names* `one` and `two`. The internal names are used as indices in the method table L_1 and in the method type table $\mathcal{L}_1$. The first method of this object returns the integer 10 and its type is `int`. The second method returns *self* (that is, the object itself) which is bound to `x` at the beginning of the object definition. The type of the method is α, the type of the object (also called *selftype*), bound just after `x`.

The type of the object x_1 is

$$\zeta(\alpha)[\mathtt{ten} : \mathtt{int}; \mathtt{clone} : \alpha]$$

The types of the methods are the translation of the method type table $\mathcal{L}_1$ using the dictionary φ_1. More precisely, it is defined as $\mathcal{L}_1 \circ \varphi_1$. These types are again parametrized over selftype α, which is bound at the beginning of the object type.

The method `clone` can be invoked using the construction x_1.`clone`. During the evaluation of this operation, the external method name `clone` is translated into the internal method name $\varphi_1(\mathtt{clone}) = \mathtt{two}$. Then, the corresponding method body $L_1(\mathtt{two}) = \mathtt{x}$ is extracted. The rest of the evaluation is standard: the type variable α is replaced by the type of the object in the method body, and the variable `x` is replaced by the object. The expression x_1.`clone` therefore evaluates into x_1. Its type is obtained by extracting the type α of the method `clone` and replacing selftype by the object type itself: it's the same as the type of x_1.

The object x_1 can actually be considered as a *prototype*, of type

$$\zeta(\alpha)[\mathtt{ten} : \mathtt{int}; \mathtt{clone} : \alpha]^{\emptyset}$$

A prototype is an object which is possibly not yet finished and can still be modified: it is possible to add a method to a prototype, to override one of its methods and to hide one of its methods. Contrary to proper objects, some of the methods of a prototype may have been given a type without being defined yet. Such a method is named an *abstract method.* The prototype x_1 has no abstract method, so the set of its abstract methods, at the upper right of its type, is empty. As in a previous calculus with prototype designed by Fisher and Mitchell [7], there is no syntactic difference between a prototype and a *"proper objects"*: they are only distinguished by their types.

[1]We use the notation ::= to name an expression. It is not part of the calculus

The construction $x_1 + (\mathtt{id} : \varsigma(\alpha)(\alpha \to \alpha))$ adds an abstract method `id` of type $\alpha \to \alpha$ (where α is selftype) to the prototype x_1. More precisely, it evaluates into a prototype x_2 which is a copy of the prototype x_1 with space reserved for one more method `id`:

$$\begin{array}{ll} x_2 ::= \zeta(\mathtt{x} : \alpha)[L_1 | \mathcal{L}_2]^{\varphi_2} & \\ \quad \text{where} & \varphi_2 ::= \mathtt{ten} \mapsto \mathtt{one}; \mathtt{clone} \mapsto \mathtt{two}; \mathtt{id} \mapsto \mathtt{three} \\ & L_1 ::= \mathtt{one} = 10; \mathtt{two} = \mathtt{x} \\ & \mathcal{L}_2 ::= \mathtt{one} : \mathtt{int}; \mathtt{two} : \alpha; \mathtt{three} : \alpha \to \alpha \end{array}$$

The dictionary has been extended and maps the method name `id` to a fresh method name `three` chosen during the reduction. The method table of the prototype is unchanged, while the method type table has been extended with the method `three`. The prototype x_2 has type

$$\zeta(\alpha)[\mathtt{ten} : \mathtt{int}; \mathtt{clone} : \alpha; \mathtt{id} : \alpha \to \alpha]^{\{\mathtt{id}\}}$$

Note that the set of abstract methods of this prototype is the singleton $\{\mathtt{id}\}$: no definition is provided for the method `id` yet.

We can now provide a definition to the method `id`, using the construction $x_2.\mathtt{id} \Leftarrow \varsigma(\mathtt{x} : \alpha)\lambda(\mathtt{y} : \alpha)\mathtt{y}$. In this expression, `x` stands for self and α stands for selftype. The expression evaluates into a new prototype x_3. This prototype has the same method type table $\mathcal{L}_2$ and dictionary φ_2 as x_2, but its method table contains the expected definition for the method `id`:

$$\begin{array}{ll} x_3 ::= \zeta(\mathtt{x} : \alpha)[L_3 | \mathcal{L}_2]^{\varphi_2} & \\ \quad \text{where} & \varphi_2 ::= \mathtt{ten} \mapsto \mathtt{one}; \mathtt{clone} \mapsto \mathtt{two}; \mathtt{id} \mapsto \mathtt{three} \\ & L_3 ::= \mathtt{one} = 10; \mathtt{two} = \mathtt{x}; \mathtt{three} = \lambda(\mathtt{y} : \alpha)\mathtt{y} \\ & \mathcal{L}_2 ::= \mathtt{one} : \mathtt{int}; \mathtt{two} : \alpha; \mathtt{three} : \alpha \to \alpha \end{array}$$

The type of the prototype x_3 is the same as the type of the prototype x_2 except there is no abstract method:

$$\zeta(\alpha)[\mathtt{ten} : \mathtt{int}; \mathtt{clone} : \alpha; \mathtt{id} : \alpha \to \alpha]^{\emptyset}$$

The same construction can be used to override a method definition: if the method is already defined, its body is replaced by the new value.

Finally, it is possible to hide a method of a prototype if this method is defined: the expression $x_3 \setminus \mathtt{ten}$ evaluates into a copy of the prototype x_3 where the method `ten` is not accessible anymore from outside the object: it is removed from the dictionary φ_4.

$$\begin{array}{ll} x_4 ::= \zeta(\mathtt{x} : \alpha)[L_3 | \mathcal{L}_2]^{\varphi_4} & \\ \quad \text{where} & \varphi_4 ::= \mathtt{clone} \mapsto \mathtt{two}; \mathtt{id} \mapsto \mathtt{three} \\ & L_3 ::= \mathtt{one} = 10; \mathtt{two} = \mathtt{x}; \mathtt{three} = \lambda(\mathtt{y} : \alpha)\mathtt{y} \\ & \mathcal{L}_2 ::= \mathtt{one} : \mathtt{int}; \mathtt{two} : \alpha; \mathtt{three} : \alpha \to \alpha \end{array}$$

The type of this prototype is the same as the prototype x_3 except for the method `ten`.

$$\zeta(\alpha)[\mathtt{clone} : \alpha; \mathtt{id} : \alpha \to \alpha]^{\emptyset}$$

$a ::= x$	Variable
$\mid \lambda(x:\tau)a$	Abstraction
$\mid a(a')$	Application
$\mid \zeta(x:\alpha)[L \mid \mathcal{L}]^{\varphi}$	Object
$\mid a + (l : \varsigma(\alpha)\tau)$	Method addition
$\mid a \setminus l$	Method hiding
$\mid a.l \Leftarrow \varsigma(x:\alpha)a'$	Method override
$\mid a.l$	Method invocation
$\tau ::= \alpha$	Type variable
$\mid \tau \to \tau'$	Function type
$\mid \zeta(\alpha)[\mathcal{L}]$	Object type
$\mid \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}$	Prototype type
$L ::= \emptyset \mid L; (l = a)$	Method table
$\mathcal{L} ::= \emptyset \mid \mathcal{L}; (l : \tau)$	Method type table
$\varphi ::= \emptyset \mid \varphi; (l \mapsto l)$	Dictionary

Figure 1: Syntax (base calculus)

Finally, a subtyping relation $\preceq$ is defined on types. A prototype can be viewed as an object, provided that all its methods are defined, by subtyping. For instance, the following relation holds:

$$\zeta(\alpha)[\texttt{ten} : \texttt{int}; \texttt{clone} : \alpha]^{\emptyset} \\ \preceq \\ \zeta(\alpha)[\texttt{ten} : \texttt{int}; \texttt{clone} : \alpha]$$

One can also hide some methods of an object type by subtyping. For instance:

$$\zeta(\alpha)[\texttt{ten} : \texttt{int}; \texttt{clone} : \alpha; \texttt{id} : \alpha \to \alpha] \\ \preceq \\ \zeta(\alpha)[\texttt{ten} : \texttt{int}; \texttt{clone} : \alpha]$$

As usual, method can be hidden this way only if the resulting type has no binary method. This is the main reason why we differentiate prototypes from proper objects: we would like the usual subtyping rules to hold for proper objects, while we would like to be able to hide any method in prototypes, which are used to define objects.

2.2 Formalization

2.2.1 Syntax

The syntax of the calculus is presented in figure 1. We have already presented each of the constructions in the examples above. It assumes three infinite sets: variables x, method names l, and type variables α. We recall that a method table L is a finite partial function from method names l to expressions a, a method type table $\mathcal{L}$ is a finite partial function from method names to types τ, and a dictionary is a finite partial function from method names to method names.

2.2.2 Static semantics

(Var)
$$\frac{(x:\tau) \in ,}{, \vdash x : \tau}$$

(Abs)
$$\frac{, ; (x:\tau') \vdash a : \tau}{, \vdash \lambda(x:\tau')a : \tau' \to \tau}$$

(App)
$$\frac{, \vdash a : \tau' \to \tau \qquad , \vdash a' : \tau'}{, \vdash a(a') : \tau}$$

(Sub)
$$\frac{, \vdash a : \tau \qquad , \vdash \tau \preceq \tau'}{, \vdash a : \tau'}$$

(Selection-Base)
$$\frac{, \vdash a : \tau \qquad \tau = \zeta(\alpha)[\mathcal{L}] \\ \mathcal{L}(l) = \tau'}{, \vdash a.l : \tau'\{\tau/\alpha\}}$$

(Extension-Base)
$$\frac{, \vdash a : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}}{, \vdash a + (l : \varsigma(\alpha)\tau) : \zeta(\alpha)[\mathcal{L}; (l:\tau)]^{\mathcal{A} \cup \{l\}}}$$

(Override-Base)
$$\frac{, \vdash a : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}} \\ , ; (\alpha); (x:\alpha) \vdash a' : \mathcal{L}(l)}{, \vdash a.l \Leftarrow \varsigma(x:\alpha)a' : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A} \setminus \{l\}}}$$

(Restriction-Base)
$$\frac{, \vdash a : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}} \qquad l \in \operatorname{dom} \mathcal{L} \setminus \mathcal{A}}{, \vdash a \setminus l : \zeta(\alpha)[\mathcal{L}|_{\operatorname{dom} \mathcal{L} \setminus \{l\}}]^{\mathcal{A}}}$$

(Proto-Base)
$$\frac{\mathcal{L}' = \mathcal{L} \circ \varphi \\ \mathcal{A} = \operatorname{dom} \mathcal{L}' \setminus \operatorname{dom}(L \circ \varphi) \\ \text{For all } l \text{ in } \operatorname{dom} L, , ; (\alpha); (x:\alpha) \vdash L(l) : \mathcal{L}(l)}{, \vdash \zeta(x:\alpha)[L \mid \mathcal{L}]^{\varphi} : \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}}$$

Figure 2: Typing rules (base calculus)

(Sub-Refl)
$$\frac{}{, \vdash \tau \preceq \tau}$$

(Sub-Arrow)
$$\frac{, \vdash \tau_1 \preceq \tau_2 \qquad , \vdash \tau_2' \preceq \tau_1'}{, \vdash \tau_1' \to \tau_1 \preceq \tau_2' \to \tau_2}$$

(Sub-Proto-Base)
$$\frac{, \vdash \zeta(\alpha)[\mathcal{L}] \preceq \zeta(\alpha)[\mathcal{L}']}{, \vdash \zeta(\alpha)[\mathcal{L}]^{\emptyset} \preceq \zeta(\alpha)[\mathcal{L}']}$$

(Sub-Object-Base)
$$\frac{\mathcal{L}' \subseteq \mathcal{L} \qquad \mathrm{co}_{\alpha}(\mathcal{L}')}{, \vdash \zeta(\alpha)[\mathcal{L}] \preceq \zeta(\alpha)[\mathcal{L}']}$$

Figure 3: Subtyping rules (base calculus)

Method addition ($l' \notin \operatorname{dom}\mathcal{L} \cup \operatorname{range}\varphi$)

$$\zeta(x:\alpha)[L|\mathcal{L}]^{\varphi} + (l : \varsigma(\alpha)\tau) \quad\longrightarrow\quad \zeta(x:\alpha)[L|(\mathcal{L};(l':\tau))]^{\varphi;(l=l')}$$

Method hiding $\quad \zeta(x:\alpha)[L|\mathcal{L}]^{\varphi} \setminus l \quad\longrightarrow\quad \zeta(x:\alpha)[L|\mathcal{L}]^{\varphi|_{\operatorname{dom}\varphi\setminus\{l\}}}$

Method override $\quad \zeta(x:\alpha)[L|\mathcal{L}]^{\varphi}.l \Leftarrow \varsigma(x:\alpha)a \quad\longrightarrow\quad \zeta(x:\alpha)[L;(\varphi(l)=a)|\mathcal{L}]^{\varphi}$

Method invocation ($v = \zeta(x:\alpha)[L|\mathcal{L}]^{\varphi}$ and $\tau = \zeta(\alpha)[\mathcal{L}\circ\varphi]$)

$$v.l \quad\longrightarrow\quad L(\varphi(l))\{\tau/\alpha\}\{v/x\}$$

Application $\quad (\lambda(x:\tau)a)(v) \quad\longrightarrow\quad a\{v/x\}$

Figure 4: Reduction rules (base calculus)

The static semantics defines a typing judgment , $\vdash a : \tau$ and a subtyping judgment , $\vdash \tau \preceq \tau'$, where an environment , is a sequence of value bindings and type variable bindings:

$$, \ ::= \emptyset \mid , ;(x:\tau) \mid , ;(\alpha)$$

The typing and subtyping rules are given in figures 2 and 3.

A type τ or an expression a are *closed* with respect to an environment , if all of their variables are bound in this environment. An environment , is closed if all of the types and view types of its range are closed with respect to , . A typing judgment , $\vdash a : \tau$ is closed if , is closed and a and τ are closed with respect to , ; a subtyping judgment , $\vdash \tau \preceq \tau'$ is closed if , is closed and τ and τ' are closed with respect to , . We assume that all judgments under discussion are closed, and that no variables are ever bound twice in an environment.

All typing rules are simple, except the rule Proto-Base. In this rule, the first condition $\mathcal{L}' = \mathcal{L}\circ\varphi$ ensures that the type $\mathcal{L}'$ of the methods in the type of the prototype is the method type table $\mathcal{L}$ translated by the dictionary φ. The second condition $\mathcal{A} = \operatorname{dom}\mathcal{L}' \setminus \operatorname{dom}(L\circ\varphi)$ ensures that the abstract methods $\mathcal{A}$ are the external methods (domain of $\mathcal{L}'$) which are not defined in the method table L. Finally, each internal method is typed in an environment where selftype α is an abstract type and self x has type α.

The object subtyping rule Sub-Object-Base requires that selftype α appears covariantly in the method types $\mathcal{L}'$ (which we note $\operatorname{co}_\alpha(\mathcal{L}')$). As usual, we say that the type variable α appears *covariantly* in a type τ if any of the following is true: α is not free in τ; τ is α; τ is $\tau_1 \to \tau_2$ and α appears contravariantly in τ_1 and covariantly in τ_2. Similarly, the type variable α appears *contravariantly* in a type τ if any of the following is true: α is not free in τ; τ is $\tau_1 \to \tau_2$ and α appears covariantly in τ_1 and contravariantly in τ_2.

2.2.3 Dynamic semantics

We give a small-step reduction semantics to our calculus. The local reduction relation $\longrightarrow$ is defined in figure 4. The rules defining this relation make use of the two standard substitution operations on values ($a\{v/x\}$) and types ($a\{\tau/\alpha\}$).

For method addition, a fresh internal method name l' is chosen. It must not be in the domain of the method type table $\mathcal{L}$ so as not to override the type of another method. It must not be in the range of the dictionary φ either. Indeed, any method name l'' such that $\varphi(l'') = l'$ would not appear in the type of the prototype before reduction and would be given type τ after reduction.

The type $\zeta(\alpha)[\mathcal{L}\circ\varphi]$ is the most general type an object $\zeta(x : \alpha)[L|\mathcal{L}]^{\varphi}$ can be given. This type is therefore substituted for selftype during method invocation.

The semantics is defined using an evaluation context F:

$$\begin{array}{rcl} F & ::= & [\_] \\ & \mid & F + (l : \varsigma(\alpha)\tau) \\ & \mid & F \setminus l \\ & \mid & F.l \Leftarrow \varsigma(x:\alpha)a \\ & \mid & F.l \\ & \mid & F(a) \\ & \mid & v(F) \end{array}$$

The local reduction relation $\longrightarrow$ is extended to a one-step evaluation relation: $a \longrightarrow a'$ iff there are expressions a_1 and a_1' such that $a = F[a_1]$, $a_1 \longrightarrow a_1'$ and $a' = F[a_1']$.

Values are defined by the following sub-grammar of expressions:

$$\begin{array}{rcll} v & ::= & x & \text{Variable} \\ & \mid & \lambda(x:\tau)a & \text{Abstraction} \\ & \mid & \zeta(x:\alpha)[L|\mathcal{L}]^{\varphi} & \text{Object} \end{array}$$

2.3 Design choices

We present and justify here some design choices common with the base calculus and the full calculus presented in next section.

2.3.1 Abstract methods

The calculus has a notion of abstract methods. This notion is not mandatory. However, in order to ensure the soundness of a calculus with method hiding and object extension, method overriding and object extension must be distinguished. Indeed, they have an incompatible semantics when they operate on a method that already exists in the object: with method overriding, already existing methods must have access to the new definition of the method; on the other hand, with object extension, a new method must be defined, different from the previous method of the same name, and the behavior of existing methods should not be altered. Rather than having these two operations as primitives as in the work of Fisher, Honsell and Mitchell [6], we have preferred to decompose object extension into the addition of an abstract method followed by the overriding of this method. We feel that these latter primitives are more atomic and orthogonal. Furthermore, they allow the definition of mutually recursive methods without resorting to tricks such as non-terminating methods (a non-terminating method body can have any type, so it can be used as a placeholder).

2.3.2 Depth subtyping

For the sake of simplicity, we have only considered width subtyping for objects in our calculus. We don't expect any difficulty with depth subtyping. Indeed, we have been careful not to introduce in the calculus any operation such as method overriding in object, which would be unsound. Depth subtyping would however make the proofs significantly longer.

An unfortunate consequence of this restriction to width subtyping is that type variables occurring in an object type are non-variant. With depth-subtyping, the definition of covariance could be updated so as to get rid of this restriction.

2.3.3 Non-deterministic dynamic semantics

The semantics we give is not deterministic. Indeed, when a method is added to a prototype, an internal method name can be arbitrarily chosen for this method. It would be easy to make this semantics deterministic by providing a choice function, associating a new method names to the set of already existing internal method names (this is what is done in the work of Riecke and Stone [18], where the methods are numbered in a consecutive way). Alternatively, one can notice that internal method names cannot be observed from outside an object, in the sense that the behavior of the object is the same whatever method names are chosen. It would therefore be possible to identify objects modulo renaming of their internal method names.

2.3.4 Typed dynamic semantics

The calculus is explicitly typed, so as to ease type checking. For the sake of subject reduction, type annotations must then be manipulated during the reduction of an expression. However, the dynamic semantics does not depend on types: types could be erased before evaluation.

2.4 Limitation of the calculus and its consequences

The simplification made in the base calculus with respect to the full calculus is essentially that no attempt is made to give a precise type to self in the base calculus: selftype is simply considered as an abstract type in method bodies. Therefore, a method cannot do anything with self or any other object of same type, except ignoring it or passing it around. In particular, a method cannot invokes another method of the same object. This limitation have different consequences that we present below.

2.4.1 Dictionaries

Dictionaries are not really necessary for this calculus. Indeed, a method cannot be invoked anymore once hidden, and could therefore just be removed from the method table. However, we preferred to introduce this non-trivial notion on the simpler calculus.

2.4.2 Covariance

The following typing rule, allowing to hide methods by subtyping even in presence of binary methods, would be sound.

$$\frac{\mathcal{L}' \subseteq \mathcal{L}}{\Gamma \vdash \zeta(\alpha)[\mathcal{L}] \preceq \zeta(\alpha)[\mathcal{L}']}$$

However, this typing rule would make type checking significantly more difficult. Indeed, consider an object x of type:

$$\tau_1 ::= \zeta(\alpha)[\texttt{clone} : \alpha; \texttt{id} : \alpha \to \alpha]$$

With the typing rule above, it would also have type:

$$\tau_2 ::= \zeta(\alpha)[\texttt{id} : \alpha \to \alpha]$$

So, the expression $x.\texttt{id}$ can be given both type $\tau_1 \to \tau_1$ and type $\tau_2 \to \tau_2$, even though these types do not have a common supertype. We have therefore chosen to only allow method hiding by subtyping when selftype does not appear covariantly in the object type (rule Sub-Object-Base). We will discuss this in more details in Section 3.4.

3. FULL CALCULUS

3.1 Informal presentation of the Calculus

We first illustrate in Section 3.1.1 the problem with method hiding and binary methods and sketch how it can be solved using views. We then describe in more details views and their interaction with binary methods in Section 3.1.2. Finally, we provide more examples in order to describe the different features of the calculus and show its expressiveness.

3.1.1 What is a Views?

Let us consider a prototype of the following type:

$$\zeta(\alpha)[\texttt{val} : \texttt{int}; \texttt{compare} : \alpha \to \texttt{bool}]^{\emptyset}$$

Suppose that we would like the method `compare` to compare the value of its argument with the value of self. Its definition would be something like: $\varsigma(\texttt{x} : \alpha)\lambda(\texttt{y} : \alpha)(\texttt{x.val} = \texttt{y.val})$ An object directly derived from the prototype would have type:

$$\zeta(\alpha)[\texttt{val} : \texttt{int}; \texttt{compare} : \alpha \to \texttt{bool}]$$

The method `val` cannot be hidden by subtyping, as α does not occur in covariant position in the type. As a consequence, the argument of the method `compare` must be an object with a method `val`, as expected by the method.

Let us now consider what happens when a prototype is derived from the prototype above by hiding the method `val`. The type of the new prototype is:

$$\zeta(\alpha)[\texttt{compare} : \alpha \to \texttt{bool}]^{\emptyset}$$

An object directly derived from the prototype has type:

$$\zeta(\alpha)[\texttt{compare} : \alpha \to \texttt{bool}]$$

According to this type, the method `compare` can be applied to an object that may not have a method `val`. This is clearly unsound. So, in order to allows both method hiding and binary methods, one need a way to keep track in the type that an object has a given method, even though this method is not listed anymore in the type. For instance, we could attach a tag, say comparable, to the object type, indicating that the object has a method val of the right type.

$$\zeta(\alpha)[\texttt{val} : \texttt{int}; \texttt{compare} : \alpha \to \texttt{bool}]^{\emptyset}_{\texttt{comparable}}$$

The tag would not be allowed to be removed as long as there is a binary method. Therefore, the object type would also have the tag:

$$\zeta(\alpha)[\texttt{compare} : \alpha \to \texttt{bool}]_{\texttt{comparable}}$$

Then the method compare can only be applied to an object which has a method val, as the type of this object is selftype which has the tag comparable.

But this is not sufficient. Indeed, the method val could added again with another type, different from the one expected by the method compare.

$$\zeta(\alpha)[\texttt{val} : \texttt{bool}; \texttt{compare} : \alpha \to \texttt{bool}]^{\emptyset}_{\texttt{comparable}}$$

The expected behavior would be the method compare to invoke the old method. Because of this, the tag cannot be simply a type feature. It must be present in some way in the dynamic semantics and must provide an access to some methods which may not be in the *main interface* of the object (that is, which are not accessible via a method invocation $a.l$). The main idea of this paper is to have a collection of *alternative interfaces* to the objects, in addition to its *main interface*. An alternative interface is selected by a name k: $a._k l$. We call these names *views*. An object type holds the set of views that can be used on the object. For instance:

$$\zeta(\alpha)[\texttt{compare} : \alpha \to \texttt{bool}]_{\{\texttt{comparable}\}}$$

A fixed type is associated to each view. This type describes the interface it gives access to. The type of the view comparable could be:

$$\zeta(\alpha)[\texttt{val} : \texttt{int}; \texttt{compare} : \alpha \to \texttt{bool}]$$

This view then gives access to a method val and a method compare. The method compare can now be defined so as not to access the method val directly, but via the view comparable.

$$\varsigma(\texttt{x} : \alpha)\lambda(\texttt{y} : \alpha)(\texttt{x}._{\texttt{comparable}}\texttt{val} = \texttt{y}._{\texttt{comparable}}\texttt{val})$$

Then, compare would continue to invoke the right method even when the method val is hidden from the main interface.

$$\zeta(\alpha)[\texttt{compare} : \alpha \to \texttt{bool}]^{\emptyset}_{\{k\}}$$

Of course, views cannot be hidden when there is a binary method. One may therefore think that we would have the same problem with views that with method names. However, contrary to method names, views are lexically scoped : we use the construction $\rho(k : t)a$ to define a new view k of type t with its scope being the expression a. This construction is similar to the one used in [20] to represent memory locations or to the ν binder of the π-calculus [14].

3.1.2 Using views

In this example, we progressively create a prototype with a binary method and another method used in the binary method, then hide this other method. We then show that the binary method can still be safely invoked.

We start with an empty object. An object is defined as in the previous calculus except for the addition of a function Φ_1 mapping views k to dictionaries φ.

$$\begin{aligned} y_1 ::= {} & \zeta(\texttt{x} : \alpha)[L_1|\mathcal{L}_1]^{\varphi_1}_{\Phi_1} \\ \text{where}\quad & L_1 ::= \emptyset \\ & \mathcal{L}_1 ::= \emptyset \\ & \varphi_1 ::= \emptyset \\ & \Phi_1 ::= \emptyset \end{aligned}$$

This object can be viewed as a prototype of type $\zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}$ where all the components $\mathcal{L}$, $\mathcal{A}$ and $\mathcal{K}$ are empty. The set $\mathcal{K}$ is the set of views of the prototype. Two methods are added:

$$(y_1 + (\texttt{compare} : \varsigma(\alpha)\alpha \to \texttt{bool})) + (\texttt{val} : \varsigma(\alpha)\texttt{int})$$

This expression evaluates into a prototype y_2:

$$\begin{aligned} y_2 ::= {} & \zeta(\texttt{x} : \alpha)[L_1|\mathcal{L}_2]^{\varphi_2}_{\Phi_1} \\ \text{where}\quad & L_1 ::= \emptyset \\ & \mathcal{L}_2 ::= \texttt{one} : \alpha \to \texttt{bool}; \texttt{two} : \texttt{int} \\ & \varphi_2 ::= \texttt{compare} \mapsto \texttt{one}; \texttt{val} \mapsto \texttt{two} \\ & \Phi_1 ::= \emptyset \end{aligned}$$

The type of this prototype is

$$\zeta(\alpha)[\texttt{compare} : \alpha \to \texttt{bool}; \texttt{val} : \texttt{int}]^{\{\texttt{compare},\texttt{val}\}}_{\emptyset}$$

We now assume that the environment contains a view k of type $\zeta(\alpha)[\texttt{compare} : \alpha \to \texttt{bool}; \texttt{val} : \texttt{int}]$. As the view has exactly the same methods as the prototype, it can be added to the prototype: $\langle y_2\rangle_k$. This expression evaluates into a prototype y_3 in which the current dictionary φ_2 is associated to the view k. Note that the type of the view is indeed the type of the methods accessible via this dictionary.

$$\begin{aligned} y_3 ::= {} & \zeta(\texttt{x} : \alpha)[L_1|\mathcal{L}_2]^{\varphi_2}_{\Phi_3} \\ \text{where}\quad & L_1 ::= \emptyset \\ & \mathcal{L}_2 ::= \texttt{one} : \alpha \to \texttt{bool}; \texttt{two} : \texttt{int} \\ & \varphi_2 ::= \texttt{compare} \mapsto \texttt{one}; \texttt{val} \mapsto \texttt{two} \\ & \Phi_3 ::= k \mapsto (\texttt{compare} \mapsto \texttt{one}; \texttt{val} \mapsto \texttt{two}) \end{aligned}$$

The type of the prototype is unchanged, except for the view:

$$\zeta(\alpha)[\texttt{compare} : \alpha \to \texttt{bool}; \texttt{val} : \texttt{int}]^{\{\texttt{compare},\texttt{val}\}}_{\{k\}}$$

When a method is defined or overridden, the new method body is typed assuming that selftype is some type α which has at least the views $\mathcal{K}$ (here $\{k\}$) of the prototype. So, we can now define a method compare that compares the value of the methods val of the object itself and of an object of the same type. The method val will be invoked via the view k that both objects are known to possess. The method definition is written:

$$y_3.\texttt{compare} \Leftarrow \varsigma(\texttt{x} : \alpha)\lambda(\texttt{y} : \alpha)\texttt{x}._k\texttt{val} = \texttt{y}._k\texttt{val}$$

It evaluates into a prototype y_4:

$$\begin{aligned} y_4 ::= {} & \zeta(\texttt{x} : \alpha)[L_4|\mathcal{L}_2]^{\varphi_2}_{\Phi_3} \\ \text{where}\quad & L_4 ::= \texttt{one} = \lambda(\texttt{y} : \alpha)(\texttt{x}._k\texttt{val} = \texttt{y}._k\texttt{val}) \\ & \mathcal{L}_2 ::= \texttt{one} : \alpha \to \texttt{bool}; \texttt{two} : \texttt{int} \\ & \varphi_2 ::= \texttt{compare} \mapsto \texttt{one}; \texttt{val} \mapsto \texttt{two} \\ & \Phi_3 ::= k \mapsto (\texttt{compare} \mapsto \texttt{one}; \texttt{val} \mapsto \texttt{two}) \end{aligned}$$

Let us finish the construction of the prototype by the definition of the method val, and the hiding of this method: $(y_4.\mathtt{val} \Leftarrow \varsigma(\mathtt{x} : \alpha)5) \setminus \mathtt{val}$. This expression evaluates into:

$$\begin{array}{lll} y_5 ::= \zeta(\mathtt{x}:\alpha)[L_5|\mathcal{L}_2]^{\varphi_5}_{\Phi_3} & & \\ \quad \text{where} & L_5 ::= & \mathtt{one} = \lambda(\mathtt{y}:\alpha)(\mathtt{x}._k\mathtt{val} = \mathtt{y}._k\mathtt{val}); \\ & & \mathtt{two} = 10 \\ & \mathcal{L}_2 ::= & \mathtt{one} : \alpha \to \mathtt{bool}; \mathtt{two} : \mathtt{int} \\ & \varphi_5 ::= & \mathtt{compare} \mapsto \mathtt{one} \\ & \Phi_3 ::= & k \mapsto (\mathtt{compare} \mapsto \mathtt{one}; \mathtt{val} \mapsto \mathtt{two}) \end{array}$$

Both methods are now defined, so the prototype has type:

$$\zeta(\alpha)[\mathtt{compare} : \alpha \to \mathtt{bool}]^{\emptyset}_{\{k\}}$$

The reader may observe that this incremental construction of a prototype suggests a way of encoding classes in this calculus. The translation of a class definition would start from the value of the parent class. First, all new methods would be added as abstract methods. Second, a view would be defined so that methods can invoke one another. Third, the methods would be defined, or overridden. Finally, the private methods would be hidden. (More generally, a class would be encoded as a function taking some initialization argument and building a prototype this way.)

Let us continue the example. As all its methods are defined, the prototype y_5 can also be considered as an object, of type

$$\zeta(\alpha)[\mathtt{compare} : \alpha \to \mathtt{bool}]_{\{k\}}$$

We define the object y_6 similarly, as the result of the evaluation of the expression $(y_4.\mathtt{val} \Leftarrow \varsigma(\mathtt{x} : \alpha)7) \setminus \mathtt{val}$.

The object y_5 has a method compare of type $\alpha \to \mathtt{bool}$ where the type α is selftype. The object y_6 of same type as y_5 can therefore be passed as argument to this method: the expression $y_5.\mathtt{compare}(y_6)$ is well-typed. The type of the object y_5 ensures that both objects y_5 and y_6 have a view k as expected by the method compare, so the evaluation of this expression will not get stuck.

During the evaluation of the expression $y_5.\mathtt{compare}(y_6)$, the expression $y_5.{}_k\mathtt{val}$ will be also evaluated. For this, the dictionary $\Phi_3(k)$ corresponding to the view is first computed. The external method name val is translated using this dictionary in the internal method name $\Phi_3(k)(\mathtt{val})$. This internal method name is used to extract the method body $L_5(\Phi_3(k)(\mathtt{val}))$. The evaluation then continues as for a direct method invocation: the object type is substituted for selftype and the object for self in the method body.

3.1.3 *Abstract views*

In the previous example, the method val was hidden from the main interface of the prototype, but it could still be accessed via the view k. So as to make this method really private, we need to find a way to prevent the use of the view k. The solution we adopt is to make the view *abstract*, using a mechanism similar to the one for abstract types [5].

We start from the function below, which could be used to define the previous objects y_5 and y_6.

$$f ::= \lambda(z : \mathtt{int})((y_4.\mathtt{val} \Leftarrow \varsigma(\mathtt{x} : \alpha)z) \setminus \mathtt{val})$$

This function has type:

$$\mathtt{int} \to \zeta(\alpha)[\mathtt{compare} : \alpha \to \mathtt{bool}]^{\emptyset}_{\{k\}}$$

We first create a package of type:

$$\tau_0 = \exists(k_0)(\mathtt{int} \to \zeta(\alpha)[\mathtt{compare} : \alpha \to \mathtt{bool}]^{\emptyset}_{\{k_0\}})$$

Note that the view k does not appear in this type anymore. The package is created by the expression below:

$$p ::= \mathtt{pack}\ f\ \mathtt{as}\ \tau_0\ \mathtt{hiding}\ \{k\}$$

We then open this package in the remainder a of the program:

$$\mathtt{open}\ p\ \mathtt{as}\ [k_1, g]\ \mathtt{in}\ a$$

In the expression a, the packaged function is named g and its type is:

$$\mathtt{int} \to \zeta(\alpha)[\mathtt{compare} : \alpha \to \mathtt{bool}]^{\emptyset}_{\{k_1\}}$$

In this type, the view k_1 is abstract: no method can be invoked via this view. But, still, if we defined two objects $y_7 ::= g(5)$ and $y_8 ::= g(7)$, it is possible to invoke the method compare of y_7 with argument y_8: $y_7.\mathtt{compare}(y_8)$. Indeed, this expression is well-typed, and furthermore the calculus ensures that any object of the same type as y_7 has a view k as expected by the method compare.

The reader may have noticed that the pack construction takes a set of views to be abstracted, not just one view. This allows to hide this set of views, for instance in the type of a prototype. Indeed, this type would otherwise rapidly be cluttered by views as each time a set of methods is added to a prototype, a view need to be also added so that the new methods can invoke one another. Furthermore, all these views could prevent to manipulate in a uniform way several prototypes with a different origin. With this construction, all prototypes with a view k_0 can be given a type of the form $\exists(k)\zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\{k_0,k\}}$ (for some $\mathcal{L}$ and $\mathcal{A}$). It would not have been possible to use subtyping instead to hide a view of a prototype in a generic way. Indeed, this would not be safe in presence of binary methods: in a prototype of type

$$\zeta(\alpha)[\mathtt{m} : \alpha \to \tau; \mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}$$

the method m may expect to be applied to an object that has at least the set of views $\mathcal{K}$. Then, it is not sound to apply it to an object that only has a subset of these views.

3.1.4 *Modeling instance variables*

Instance variable can be modeled in the calculus using two methods, as usual: the method get returns the value of the instance variable and the method set set this value:

$$\begin{array}{lll} \zeta(\mathtt{x}:\alpha)[L|\mathcal{L}]^{\varphi}_{\Phi} & & \\ \quad \text{where} & L ::= & \mathtt{one} = 5; \\ & & \mathtt{two} = \lambda(\mathtt{y}:\mathtt{int})(\mathtt{x}._k\mathtt{get} \Leftarrow \varsigma(\mathtt{x}' : \alpha')\mathtt{y}) \\ & \mathcal{L} ::= & \mathtt{one} : \mathtt{int}; \mathtt{two} : \mathtt{int} \to \alpha \\ & \varphi ::= & \mathtt{get} \mapsto \mathtt{one}; \mathtt{set} \mapsto \mathtt{two} \\ & \Phi ::= & k \mapsto (\mathtt{get} \mapsto \mathtt{one}; \mathtt{set} \mapsto \mathtt{two}) \end{array}$$

The method set expect an argument y and override the body of the method get, via the view k, with a new body returning the value of y. The type of this object is:

$$\zeta(\alpha)[\mathtt{get} : \mathtt{int}; \mathtt{set} : \mathtt{int} \to \alpha]_{\{k\}}$$

The calculus does not include any operator allowing to override a method via the main interface of an object. One reason for this is that, though such an operation would be sound with the current subtyping rules, it would be unsound with depth subtyping. On the other hand, overriding via a view would remain safe, as a view has a fixed type.

3.1.5 Objects without binary methods

In the calculus, views are needed for typing binary methods, and also in prototypes for defining mutually recursive methods. But, looking at the previous examples, one may think that views would pervasively clutter all object types even when they are not needed anymore for manipulating the object. However, the subtyping rules always allows to remove all views from an object type without binary method. So, for instance, the following subtyping relation holds:

$$\begin{array}{c}\zeta(\alpha)[\mathtt{get} : \mathtt{int}; \mathtt{set} : \mathtt{int} \to \alpha]_{\{k\}} \\ \preceq \\ \zeta(\alpha)[\mathtt{get} : \mathtt{int}; \mathtt{set} : \mathtt{int} \to \alpha]_{\emptyset}\end{array}$$

The subtyping rules for an object without views is the same as in the previous calculus and are standard.

It is desirable to consider self as a regular object, and in particular to be able to apply it to a function defined outside a method body. However, self is not given an object type, but rather it is only assumed that it at least a given set of views. So, it cannot be directly applied to a function such as the one below:

$$f ::= \lambda(\mathtt{y} : \zeta(\alpha)[\mathtt{val} : \mathtt{int}]_{\emptyset})\mathtt{y.val}$$

This is nevertheless possible in an indirect way when the object has no binary method. For instance, let us consider the prototype below.

$$\begin{array}{lll} z ::= \zeta(\mathtt{x} : \alpha)[L|\mathcal{L}]^{\varphi}_{\Phi} & & \\ \quad\text{where} & L ::= & \mathtt{one} = 5; \\ & \mathcal{L} ::= & \mathtt{one} : \mathtt{int}; \mathtt{two} : \mathtt{int} \\ & \varphi ::= & \mathtt{val} \mapsto \mathtt{one}; \mathtt{use} \mapsto \mathtt{two} \\ & \Phi ::= & k \mapsto (\mathtt{val} \mapsto \mathtt{one}; \mathtt{use} \mapsto \mathtt{two}) \end{array}$$

We assume that the view k has type

$$\zeta(\alpha)[\mathtt{val} : \mathtt{int}; \mathtt{use} : \mathtt{int}]$$

We want to define the method `use` of the prototype z so that it apply the function f to self. As we said just above, inside the body of the method `use` we only know that self has a view k, so it is not possible to directly apply the function to self as follows:

$$z.\mathtt{use} \Leftarrow \varsigma(\mathtt{x} : \alpha)f(\mathtt{x})$$

This would actually not even be sound as the method `val` could be latter hidden from the main interface of the prototype (and therefore would not be bound in the main dictionary anymore). However, a subtyping rule allows to consider self `x` as having the type $\zeta(\alpha)[\emptyset]_{\{k\}}$. (This is only possible because selftype only occurs in covariant position in the type of the view k.) Then, the construction $\mathtt{x}|_k$ can be used to replace the main dictionary of the object `x` by the dictionary associated to the view k. The expression $\mathtt{x}|_k$ has type $\zeta(\alpha)[\mathtt{val} : \mathtt{int}; \mathtt{use} : \mathtt{int}]_{\{k\}}$. By subtyping, it has also type $\zeta(\alpha)[\mathtt{val} : \mathtt{int}]_{\emptyset}$, and so we can apply the function f to this expression:

$$z.\mathtt{use} \Leftarrow \varsigma(\mathtt{x} : \alpha)f(\mathtt{x}|_k)$$

3.1.6 Friend Functions

A friend function is a function that has a privileged access to objects of a class. The existence of such privileged access is of particular importance for privacy. Indeed, if for instance two objects needed to interact between each other, all methods necessary for this interaction would otherwise have to remain public. The usual trick to encode friend functions in a structural typing setting is to use a method `repr` that returns the whole internal state of the object [15]. So that only friend functions can access this state, the type of this method is hidden using an abstract type. However, this technique defeats our goal of being able to hide arbitrary methods. Indeed, if the method is hidden in a subclass, the friend function will not be able to manipulate objects of this subclass. Another possibility is to embed the class and its friend functions into a module than does not export the class, but only a constructor for the objects of the class [9]. Any method of the class can be hidden in the type of the constructor by making it partially abstract. However, this technique precludes any further subclassing. Thus, both these solutions are unsatisfactory. We show how public views can be used to implement friend functions without any of these limitations. We have actually presented all the necessary ingredients. We first define a class c taking an initialization argument `y` and returning a prototype with a method `val` whose value is the value of the class argument.

$$\begin{array}{l} c ::= \lambda(\mathtt{y} : \mathtt{int}) \\ \qquad \langle\zeta(\mathtt{x} : \alpha)[\emptyset|\emptyset]^{\emptyset}_{\emptyset} + (\mathtt{val} : \varsigma(\alpha)\mathtt{int})\rangle_k.\mathtt{val} \Leftarrow \varsigma(\mathtt{x} : \alpha)\mathtt{y} \end{array}$$

The prototype is defined using the following view:

$$k : \zeta(\alpha)[\mathtt{val} : \mathtt{int}]$$

The class has type:

$$\mathtt{int} \to \zeta(\alpha)[\mathtt{val} : \mathtt{int}]^{\emptyset}_{\{k\}}$$

We now define a function f invoking the method `val` of its argument via the view k.

$$f ::= \lambda(\mathtt{x} : \zeta(\alpha)[\emptyset]_{\{k\}})\mathtt{x}._k\mathtt{val}$$

We want to prevent the access to the method `val` of the class c from anywhere but the function f.

First, a wrapper takes care of hiding the method `val` from the main interface of the prototype returned by the class c. (It is still accessible via the view k.)

$$c' ::= \lambda(\mathtt{z} : \mathtt{int})(c(\mathtt{z})) \setminus \mathtt{val}$$

Then, the class and the function are both put in an object (this object can be viewed as a record, or a module).

$$\begin{array}{ll} m ::= & (\zeta(\mathtt{x} : \alpha)[\emptyset|\emptyset]^{\emptyset}_{\emptyset} \\ & \quad + (\mathtt{c} : \varsigma(\alpha)(\mathtt{int} \to \zeta(\alpha)[\emptyset]^{\emptyset}_{\{k\}})) \\ & \quad .\mathtt{c} \Leftarrow \varsigma(\mathtt{x} : \alpha)c') \\ & \quad + (\mathtt{f} : \varsigma(\alpha)(\zeta(\alpha)[\emptyset]_{\{k\}} \to \mathtt{int})) \\ & \quad .\mathtt{f} \Leftarrow \varsigma(\mathtt{x} : \alpha)f \end{array}$$

The view is then hidden by an abstract view k_0:

$$p ::= \mathtt{pack}\ m\ \mathtt{as}\ \tau_0\ \mathtt{hiding}\ \{k\}$$

where

$$\begin{array}{c} \tau_0 ::= \\ \exists(k_0)\zeta(\alpha)[\mathtt{c} : \mathtt{int} \to \zeta(\alpha)[\emptyset]^{\emptyset}_{\{k_0\}}; \mathtt{f} : \zeta(\alpha)[\emptyset]_{\{k_0\}} \to \mathtt{int}]_{\emptyset} \end{array}$$

$a ::= x$	Variable
$\mid \lambda(x : \tau)a$	Abstraction
$\mid a(a')$	Application
$\mid \zeta(x : \alpha)[L \mid \mathcal{L}]^{\varphi}_{\Phi}$	Object
$\mid a + (l : \varsigma(\alpha)\tau)$	Method addition
$\mid a \setminus l$	Method hiding
$\mid a.l$	Method selection
$\mid a._{k}l$	Method selection in view
$\mid a.l \Leftarrow \varsigma(x : \alpha)a'$	Method override
$\mid a._{k}l \Leftarrow \varsigma(x : \alpha)a'$	Method override in view
$\mid \rho(k : t)a$	View binding
$\mid \langle a \rangle_{k}$	View addition
$\mid a\|_{k}$	View replacement
$\mid$ `pack` a `as` τ `hiding` $\mathcal{K}$	Packing
$\mid$ `open` a `as` $[k, x]$ `in` a'	Unpacking
$\tau ::= \alpha$	Type variable
$\mid \tau \to \tau'$	Function type
$\mid t_{\mathcal{K}}$	Object type
$\mid t^{\mathcal{A}}_{\mathcal{K}}$	Prototype type
$\mid \exists(k)\tau$	View abstraction
$t ::= \zeta(\alpha)[\mathcal{L}]$	Interface type
$L ::= \emptyset \mid L; (l = a)$	Method table
$\mathcal{L} ::= \emptyset \mid \mathcal{L}; (l : \tau)$	Method type table
$\varphi ::= \emptyset \mid \varphi; (l \mapsto l)$	Dictionary
$\Phi ::= \emptyset \mid \Phi; (k \mapsto \varphi)$	Dictionary table
$\mathcal{K} ::= \emptyset \mid \{k, \ldots, k\}$	View set

Figure 5: Syntax

Finally, the class and the function are made available to the remainder a of the program:

$$\texttt{open}\ m\ \texttt{as}\ [k_1, \texttt{m}_1]\ \texttt{in}\ a$$

The type of the class $\texttt{m}_1\texttt{.c}$ is

$$\texttt{int} \to \zeta(\alpha)[\emptyset]^{\emptyset}_{\{k_1\}}$$

The type of the function $\texttt{m}_1\texttt{.f}$ is

$$\zeta(\alpha)[\emptyset]_{\{k_1\}} \to \texttt{int}$$

The type system ensures that only objects derived from the class can have the view k_0. As views cannot be removed from objects, all these object have a view k, as expected by the function $\texttt{m}_1\texttt{.f}$, so it would be sound to apply them to this function.

3.2 Formalization

(Sub-Refl)

$$\frac{}{,\ \vdash \tau \preceq \tau}$$

(Sub-Arrow)

$$\frac{,\ \vdash \tau_1 \preceq \tau_2 \qquad ,\ \vdash \tau_2' \preceq \tau_1'}{,\ \vdash \tau_1' \to \tau_1 \preceq \tau_2' \to \tau_2}$$

(Sub-Proto)

$$\frac{,\ \vdash \zeta(\alpha)[\mathcal{L}]_{\mathcal{K}} \preceq \zeta(\alpha)[\mathcal{L}']_{\mathcal{K}'}}{,\ \vdash \zeta(\alpha)[\mathcal{L}]^{\emptyset}_{\mathcal{K}} \preceq \zeta(\alpha)[\mathcal{L}']_{\mathcal{K}'}}$$

(Sub-Object)

$$\frac{\mathcal{L}' \subseteq \mathcal{L} \qquad \mathrm{co}_{\alpha}(\mathcal{L}') \qquad \mathcal{K}' \subseteq \mathcal{K} \\ \forall k \in \mathcal{K}' \cdot (k : \zeta(\alpha)[\mathcal{L}'']) \in\ , \ \wedge \mathrm{co}_{\alpha}(\mathcal{L}'')}{,\ \vdash \zeta(\alpha)[\mathcal{L}]_{\mathcal{K}} \preceq \zeta(\alpha)[\mathcal{L}']_{\mathcal{K}'}}$$

(Sub-Match)

$$\frac{,\ \vdash \alpha <\# \mathcal{K} \\ \forall k \in \mathcal{K} \cdot (k : \zeta(\alpha)[\mathcal{L}'']) \in\ , \ \wedge \mathrm{co}_{\alpha}(\mathcal{L}'')}{,\ \vdash \alpha \preceq \zeta(\alpha')[\emptyset]_{\mathcal{K}}}$$

Figure 7: Subtyping rules

(Match-Var)

$$\frac{(\alpha <\# \mathcal{K}) \in\ , \qquad \mathcal{K}' \subseteq \mathcal{K}}{,\ \vdash \alpha <\# \mathcal{K}'}$$

(Match-Obj)

$$\frac{\mathcal{K}' \subseteq \mathcal{K}}{,\ \vdash \zeta(\alpha)[\mathcal{L}]_{\mathcal{K}} <\# \mathcal{K}'}$$

Figure 8: Matching rules

The calculus is an extension of the calculus presented in the first section. The formalization reuses the same notations.

3.2.1 *Syntax*

The syntax of the calculus is presented in figure 5. It assumes an infinite set of views k. We recall that a dictionary table Φ is a finite partial function from views k to dictionaries φ.

3.2.2 *Static semantics*

The static semantics defines a typing judgment $,\ \vdash a : \tau$ and a subtyping judgment $,\ \vdash \tau \preceq \tau'$, where an environment $,$ is a sequence of bindings. An environment contains both value bindings and view bindings. It also contains what we call *matching hypothesis*, that is the hypothesis that a type α has at least the set of views $\mathcal{K}$ (this is not the same notion of matching as the one introduced by Kim Bruce [4], but the two notions have similarities: types that match a given set of views have a similar shape, but are not necessarily subtypes of a same type).

$,\ ::= \emptyset$	Empty environment
$\mid\ , ; (x : \tau)$	Value binding
$\mid\ , ; (\alpha <\# \mathcal{K})$	Matching hypothesis
$\mid\ , ; (k : t)$	View binding
$\mid\ , ; (k)$	Abstract view binding

The typing rules are given in figures 6, 7, and 8. As in the previous calculus (Section 2.2.2), all judgment are assumed to be closed and no variable are ever bound twice in an environment.

(Var)
$$\frac{(x:\tau)\in\,,}{,\ \vdash x:\tau}$$

(Abs)
$$\frac{,\ ;(x:\tau')\vdash a:\tau}{,\ \vdash \lambda(x:\tau')a:\tau'\to\tau}$$

(App)
$$\frac{,\ \vdash a:\tau'\to\tau \qquad ,\ \vdash a':\tau'}{,\ \vdash a(a'):\tau}$$

(Sub)
$$\frac{,\ \vdash a:\tau \qquad ,\ \vdash \tau\preceq\tau'}{,\ \vdash a:\tau'}$$

(Proto)
$$\frac{\begin{array}{ll}\mathcal{L}'=\mathcal{L}\circ\varphi & \mathcal{A}=\mathrm{dom}\,\mathcal{L}'\setminus\mathrm{dom}(L\circ\varphi)\\ \mathcal{K}=\mathrm{dom}\,\Phi & \mathrm{dom}\,\mathcal{L}\subseteq\mathrm{dom}\,L\cup\mathrm{range}\,\varphi\\ \multicolumn{2}{l}{\text{For all } l \text{ in dom } L,\ ,\ ;(\alpha<\#\,\mathcal{K});(x:\alpha)\vdash L(l):\mathcal{L}(l)}\\ \multicolumn{2}{l}{\forall k\in\mathrm{dom}\,\Phi\cdot(k:\zeta(\alpha)[\mathcal{L}''])\in\,,\ \wedge\ \mathcal{L}''=\mathcal{L}\circ\Phi(k)}\end{array}}{,\ \vdash \zeta(x:\alpha)[L|\mathcal{L}]^{\varphi}_{\Phi}:\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}}}$$

(Extension)
$$\frac{,\ \vdash a:\zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}}{,\ \vdash a+(l:\varsigma(\alpha)\tau):\zeta(\alpha)[\mathcal{L};(l:\tau)]^{\mathcal{A}\cup\{l\}}_{\mathcal{K}}}$$

(Restriction)
$$\frac{,\ \vdash a:\zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}} \qquad l\in\mathrm{dom}\,\mathcal{L}\setminus\mathcal{A}}{,\ \vdash a\setminus l:\zeta(\alpha)[\mathcal{L}|_{\mathrm{dom}\,\mathcal{L}\setminus\{l\}}]^{\mathcal{A}}_{\mathcal{K}}}$$

(Selection)
$$\frac{\begin{array}{c},\ \vdash a:\tau \qquad \tau=\zeta(\alpha)[\mathcal{L}]_{\mathcal{K}}\\ (l:\tau')\in\mathcal{L}\end{array}}{,\ \vdash a.l:\tau'\{\tau/\alpha\}}$$

(View-Selection)
$$\frac{\begin{array}{c},\ \vdash a:\tau \qquad ,\ \vdash \tau<\#\,\mathcal{K}\\ k\in\mathcal{K} \qquad (k:\zeta(\alpha)[\mathcal{L}'])\in\,,\\ (l:\tau')\in\mathcal{L}'\end{array}}{,\ \vdash a._k l:\tau'\{\tau/\alpha\}}$$

(Override)
$$\frac{\begin{array}{c},\ \vdash a:\zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}\\ ,\ ;(\alpha<\#\,\mathcal{K});(x:\alpha)\vdash a':\mathcal{L}(l)\end{array}}{,\ \vdash a.l\Leftarrow\varsigma(x:\alpha)a':\zeta(\alpha)[\mathcal{L}]^{\mathcal{A}\setminus\{l\}}_{\mathcal{K}}}$$

(View-Override)
$$\frac{\begin{array}{c},\ \vdash a:\tau \qquad ,\ \vdash \tau<\#\,\mathcal{K}\\ k\in\mathcal{K} \qquad (k:\zeta(\alpha)[\mathcal{L}'])\in\,,\\ ,\ ;(\alpha<\#\,\mathcal{K});(x:\alpha)\vdash a':\mathcal{L}'(l)\end{array}}{,\ \vdash a._k l\Leftarrow\varsigma(x:\alpha)a':\tau}$$

(View-Abs)
$$\frac{,\ ;(k:t)\vdash a:\tau}{,\ \vdash \rho(k:t)a:\tau}$$

(View-Capture)
$$\frac{\begin{array}{c},\ \vdash a:\zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}\\ (k:\zeta(\alpha)[\mathcal{L}])\in\,,\end{array}}{,\ \vdash \langle a\rangle_k:\zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}\cup\{k\}}}$$

(View-Replace)
$$\frac{\begin{array}{c},\ \vdash a:\zeta(\alpha)[\mathcal{L}]_{\mathcal{K}}\\ k\in\mathcal{K}\\ (k:\zeta(\alpha)[\mathcal{L}'])\in\,,\end{array}}{,\ \vdash a|_k:\zeta(\alpha)[\mathcal{L}']_{\mathcal{K}}}$$

(Pack)
$$\frac{,\ \vdash a:\tau\{\mathcal{K}/k\}}{,\ \vdash \texttt{pack}\ a\ \texttt{as}\ \exists(k)\tau\ \texttt{hiding}\ \mathcal{K}:\exists(k)\tau}$$

(Open)
$$\frac{\begin{array}{c},\ \vdash a':\exists(k)\tau'\\ ,\ ;(k);(x:\tau')\vdash a:\tau\end{array}}{,\ \vdash \texttt{open}\ a'\ \texttt{as}\ [k,x]\ \texttt{in}\ a:\tau}$$

Figure 6: Typing rules

Method addition
$(l' \notin \mathrm{dom}\,\mathcal{L} \cup \mathrm{range}\,\varphi \cup \bigcup_{k \in \mathrm{dom}\,\Phi} \mathrm{range}\,\Phi(k))$

$$\zeta(x:\alpha)[L|\mathcal{L}]^{\varphi}_{\Phi} + (l : \varsigma(\alpha)\tau) \quad \longrightarrow \quad \zeta(x:\alpha)[L|(\mathcal{L};(l':\tau))]^{\varphi;(l=l')}_{\Phi}$$

Method hiding

$$\zeta(x:\alpha)[L|\mathcal{L}]^{\varphi}_{\Phi} \setminus l \quad \longrightarrow \quad \zeta(x:\alpha)[L|\mathcal{L}]^{\varphi|_{\mathrm{dom}\,\varphi \setminus \{l\}}}_{\Phi}$$

Override

$$\zeta(x:\alpha)[L|\mathcal{L}]^{\varphi}_{\Phi}.l \Leftarrow \varsigma(x:\alpha)a \quad \longrightarrow \quad \zeta(x:\alpha)[L;(\varphi(l)=a)|\mathcal{L}]^{\varphi}_{\Phi}$$

$$\zeta(x:\alpha)[L|\mathcal{L}]^{\varphi}_{\Phi}.{}_k l \Leftarrow \varsigma(x:\alpha)a \quad \longrightarrow \quad \zeta(x:\alpha)[L;(\Phi(k)(l)=a)|\mathcal{L}]^{\varphi}_{\Phi}$$

Method invocation
$(v = \zeta(x:\alpha)[L|\mathcal{L}]^{\varphi}_{\Phi}$ and $\tau = \zeta(\alpha)[\mathcal{L} \circ \varphi]_{\mathrm{dom}\,\Phi})$

$$\begin{aligned} v.l &\longrightarrow L(\varphi(l))\{\tau/\alpha\}\{v/x\} \\ v.{}_k l &\longrightarrow L(\Phi(k)(l))\{\tau/\alpha\}\{v/x\} \end{aligned}$$

View manipulations

$$\begin{aligned} \langle \zeta(x:\alpha)[L|\mathcal{L}]^{\varphi}_{\Phi} \rangle_k &\longrightarrow \zeta(x:\alpha)[L|\mathcal{L}]^{\varphi}_{\Phi;(k=\varphi)} \\ \zeta(x:\alpha)[L|\mathcal{L}]^{\varphi}_{\Phi}\,|_k &\longrightarrow \zeta(x:\alpha)[L|\mathcal{L}]^{\Phi(k)}_{\Phi} \end{aligned}$$

Application

$$(\lambda(x:\tau)a)(v) \quad \longrightarrow \quad a\{v/x\}$$

Unpacking ($v =$ `pack` v' `as` $\exists(k)\tau$ `hiding` $\mathcal{K}$)

$$\texttt{open}\ v\ \texttt{as}\ [k,x]\ \texttt{in}\ a \quad \longrightarrow \quad a\{\mathcal{K}/k\}\{v'/x\}$$

View bindings ($F \neq [\_]$)

$$F[\rho(k:t)a] \quad \longrightarrow \quad \rho(k:t)F[a]$$

Figure 9: Reduction rules

We only present the most complex rules. To be able to override a method l via a view k in an object a of type τ (rule View-Override), the object must have at least a set of view $\mathcal{K}$ containing the view k, the view must be bound in the environment to a type $\zeta(\alpha)[\mathcal{L}']$ and the type of the method body, assuming that self has at least the method $\mathcal{K}$, must be the type of the method k in $\mathcal{L}'$. The rule View-Selection is modeled after rule View-Override (for accessing the type of the view) and rule Selection. In the rule Proto, the first condition $\mathcal{L}' = \mathcal{L} \circ \varphi$ ensures that the type $\mathcal{L}'$ of the methods in the type of the prototype is the method type table $\mathcal{L}$ translated by the dictionary φ. The second condition $\mathcal{A} = \mathrm{dom}\,\mathcal{L}' \setminus \mathrm{dom}(L \circ \varphi)$ asserts that the abstract methods $\mathcal{A}$ are the external methods (domain of $\mathcal{L}'$) which are not defined in the method table L. The third condition $\mathcal{K} = \mathrm{dom}\,\Phi$ asserts that the views $\mathcal{K}$ of the prototype are the domain of the function Φ mapping views to dictionaries. The fourth condition $\mathrm{dom}\,\mathcal{L} \subseteq \mathrm{dom}\,L \cup \mathrm{range}\,\varphi$ together with the second condition ensures that the equality $\mathrm{dom}\,\mathcal{L} = \mathrm{dom}\,L$ holds for a proper object (where $\mathcal{A} = \emptyset$). This then ensures that the dictionary can be soundly replaced in an object by any other dictionary (in rule View-Replace). Fifth, each internal method is typed in an environment where selftype α is an abstract type and self x has type α. Finally, all views of the prototype must be defined in the environment and their type must match the internal type of the prototype, as translated by the dictionary associated to the view ($\mathcal{L}'' = \mathcal{L} \circ \Phi(k)$). This latter condition together with the fourth condition ensures that all methods that can be accessed via the view (rule View-Selection) are indeed defined.

The rule Sub-Object allows method and view hiding in object types, as long as selftype is covariant in the resulting object type and in the types of the views associated to the object type. The rule Sub-Match is similar and formalizes the idea that any object type that has at least a set of views $\mathcal{K}$ is always a subtype of $\zeta(\alpha')[\emptyset]_{\mathcal{K}}$, provided that this latter type satisfies the same restrictions as the resulting type of rule Sub-Object.

3.2.3 *Dynamic semantics*

The local reduction relation $\longrightarrow$ is defined in figure 9. In addition to the substitutions on values and types, the rules defining this relation make use of a third substitution operation that replaces a view by a set of views. This operation is such that:

$$\begin{aligned} \mathcal{K}\{\mathcal{K}'/k\} &= \mathcal{K} \setminus \{k\} \cup \mathcal{K}' \text{ if } k \in \mathcal{K} \\ &= \mathcal{K} \text{ otherwise.} \end{aligned}$$

and is otherwise similar to the other substitutions.

For method addition, a fresh internal method name l is chosen. It must not be in the domain of the method type table $\mathcal{L}$ so that not to override the type of another method, and it must not be in the range of the dictionary φ, or any of the dictionaries $\Phi(k)$ so that only the method l be added to the main interface (and not also all methods whose image by φ is l), and so that the alternative interfaces remain unchanged. The rule for an expression $F[\rho(k:t)a]$ pushes view bindings outwards, allowing the interaction of of the expression a with the context $F[\_]$.

The semantics is defined using the contexts E and F. The context E lists the views introduced during the evaluation while the context F indicates where the evaluation takes place.

$$\begin{array}{llll} E ::= [\_] & & F ::= [\_] & \\ \mid \rho(k : t)E & & & \mid F + (l : \varsigma(\alpha)\tau) \\ & & & \mid F \setminus l \\ & & & \mid F.l \Leftarrow \varsigma(x : \alpha)a \\ & & & \mid F._k l \Leftarrow \varsigma(x : \alpha)a \\ & & & \mid F.l \\ & & & \mid F._k l \\ & & & \mid \langle F \rangle_k \\ & & & \mid F|_k \\ & & & \mid F(a) \\ & & & \mid v(F) \\ & & & \mid \mathtt{pack}\ F\ \mathtt{as}\ \tau\ \mathtt{hiding}\ \mathcal{K} \\ & & & \mid \mathtt{open}\ F\ \mathtt{as}\ [k, x]\ \mathtt{in}\ a \end{array}$$

Evaluation contexts are contexts of the form $E[F[\_]]$: the local reduction relation $\longrightarrow$ is extended to a one-step evaluation relation: $a \longrightarrow a'$ iff there are expressions a_1 and a'_1 such that $a = E[F[a_1]]$, $a_1 \longrightarrow a'_1$ and $a' = E[F[a'_1]]$.

Values are defined by the following sub-grammar of expressions:

$$\begin{array}{lr} v ::= x & \text{Variable} \\ \mid \lambda(x : \tau)a & \text{Abstraction} \\ \mid \zeta(x : \alpha)[L|\mathcal{L}]^{\varphi}_{\Phi} & \text{Object} \\ \mid \mathtt{pack}\ v\ \mathtt{as}\ \tau\ \mathtt{hiding}\ \mathcal{K} & \text{View abstraction} \end{array}$$

In addition to the rule for method addition, the rule pushing view bindings outwards introduces another source of non-determinism, as several contexts F may be possible (as in[20]). However, it is easy to convince oneself that the choice of the context will not modify the result of the evaluation.

3.3 Soundness

We have proved the following soundness results:

Theorem 1 (Subject reduction). *If* , $\vdash a : \tau$ *and* $a \longrightarrow a'$*, then* , $\vdash a' : \tau$.

Theorem 2 (Progress). *If* $\vdash a : \tau$ *then* a *is* $E[v]$ *for some value* v *or* $a \longrightarrow a'$ *for some expression* a'.

The proofs are standard. The proof of subject reduction is quite lengthy, as most reduction rules involve objects and the typing rule for objects (rule Proto) is large. But there is no specific difficulty.

3.4 Typing issues

3.4.1 Subtyping

The calculus has been designed so as to make type checking straightforward. A consequence of this is that the subtyping rule Sub-Object is more restrictive than what type soundness alone would have required. Indeed, while this would be sound, the rule does not allow the hiding of methods when selftype is not covariant in a view of the object. If this restriction was removed, the calculus would not have principal types. Indeed, let us consider for instance an object x of type

$$\tau_0 = \zeta(\alpha)[n : \mathtt{int}]_{\{k\}}$$

where the view k, has type $\zeta(\alpha)[m : \alpha \to \alpha]$. Then $x._k m$ has type $\tau_0 \to \tau_0$. If subtyping were allowed in this case, x would also have type

$$\tau_1 = \zeta(\alpha)[\,]_{\{k\}}.$$

Then, $x._k m$ would also have type $\tau_1 \to \tau_1$, which is not comparable with type $\tau_0 \to \tau_0$. This difficulty could be avoided using higher-order types: indeed, the type of $x._k m$ could be $\alpha \to \alpha$ for all α such that $\tau_0 \preceq \alpha \preceq \tau_1$. Hiding arbitrary method in a proper object with a binary method would also be safe, but raises the same difficulty, as previously mentionned.

3.4.2 Type inference

We believe that the typing rules could be easily adapted in order to make type inference possible, using row variables [19] in a similar way to Objective Caml [17, 16]. Of course, subtyping would have to be explicit, but according to our experience with Objective Caml, this is not a problem in practice. Furthermore, this would allow us to get rid of the restrictions expose in the paragraph above. The type of an object would be composed of two possibly extensible rows: one for methods, the other one for views. We expect that the typing of self would then not require any special rule. Indeed, selftype could be modeled as an object type with no method and an extensible row of views.

4. PROBLEMS WITH MULTIPLE INHERITANCE

While this calculus can be used for a class-based language with single inheritance, it unfortunately exhibits an anomaly that makes problematic its use as a base for a language with multiple inheritance. Indeed, abstraction can be broken under certain circumstance.

Indeed, let us consider an object with a view k. This view can be hidden from the type of the object using an abstract view. One could think that there is then no way to use this view of the object from outside anymore. In particular, one would expect that the methods that can only be access via this view cannot be redefined, and that the dictionary associated to this view in the object cannot be changed. However, if the view is still available in the environment, nothing prevent us from adding it again to the object. This will change the dictionary associated to the view in the object: it will become the same as the dictionary of the primary interface of the object. And methods can then be redefined using this view.

In a language with single inheritance, the scope of the views can be controlled so as to avoid this problem. With multiple inheritance, however, if a class inherits twice from another class (either directly or indirectly), it will inherit from its views twice. So, if a view have been hidden on only one of the inheritance paths, it will still be visible in the class.

5. RELATED WORK

Riecke and Stone [18] have designed an object calculus with object extension where methods can be arbitrarily hidden. This is achieved using subsumption and by a privileged access to other methods of self from a method definition. Their calculus does not handle binary methods. Our calculus is strongly inspired by this work. Indeed, we reuse the idea of dictionaries for decoupling the internal naming from the external naming of methods. Apart from its strong connection, the calculi use completely different mechanisms to achieve their goal.

Moby [8] is an experimental object-oriented language. It aims at providing a good interaction between classes and modules. Like our language, it allows arbitrary method hiding in classes. It is based on a variant of the first-order version of Riecke and Stone calculus with stateful objects. As a consequence, it does have the notion of selftype. An extension of Moby with inheritance-based subtyping has been recently proposed by Fisher and Reppy [10]. The calculus they have developed to validate their design (XMOC) is class-based. Classes play in this calculus a role similar to views in our calculus. They propose in this paper a solution for friend functions (written in Extended Moby) which is very close to the one of Section 3.1.6.

Several calculi combining object extension with object subsumption have been proposed, starting with the work of Fisher and Mitchell [7]. They ensure soundness by having two kind of object types, prototypes and proper objects, as our calculus. They usually do not allow method hiding in prototypes. This is the case for the calculus presented by Gianantonio, Honsell and Liquori in [13], as well as the one proposed by Bono, Bugliesi, Liquori and Dezani-Ciancaglini in [1]. In the calculus presented by Bono and Fisher in [2], it is possible to hide the type of any methods of a prototype. This is achieved by wrapping prototypes within existentials. Our calculus use the same idea to hide views (this idea is initially due to Pierce and Turner [15]). However, the methods are not completely hidden: their names remain visible and the prototype cannot be extended with new methods of the same name.

A need for context-dependent behavior of objects has emerged in the context of database languages: depending on the context, an object would play different roles, while retaining its identity. This notion of *roles* has been formalized by Ghelli and Palmerini [12]. They present a first-order calculus with roles. It appears that views are similar to roles though they have been introduced for a completely different purpose. Contrary to our calculus, however, each method of an object belongs to exactly one role. Another difference is that the roles form a hierarchy, and at the invocation of a method l, the method selected is the method associated to l in the minimum super-role of the current role.

Flatt, Krishnamurthi and Felleisen [11] make use of views to define powerful mixins (a mixin is a function from classes to classes). However, their mixin language does not have selftype.

6. CONCLUSION

This paper presents an object-oriented calculus allowing arbitrary hiding of methods in prototypes. This calculus also has binary methods and can be used to encode friend functions. We believe that this is the first calculus smoothly combining all these features. This demonstrates that good modularity properties of an object-oriented language does not necessarily come at the expense of its expressiveness.

Views are a key ingredient for the soundness of our calculus. They are also an interesting feature for an object-oriented language. Indeed, they bring more flexibility by allowing an object to simultaneously possess several distinct interfaces. We therefore believe that this notion could be used fruitfully for other purposes.

Our calculus does not currently handle multiple inheritance. We hope that multiple inheritance is possible even though this might requires a significantly different semantics. This is another important direction for future work.

7. REFERENCES

[1] V. Bono, M. Bugliesi, L. Liquori, and M. Dezani-Ciancaglini. Subtyping constraint for incomplete objects. In *Proceedings of TAPSOFT/CAAP*, number 1214 in LNCS, pages 465–477. Springer-Verlag, 1997.

[2] V. Bono and K. Fisher. An imperative, first-order calculus with object extension. In *ECOOP*, pages 462–497, 1998.

[3] K. B. Bruce, L. Cardelli, G. Castagna, T. H. O. Group, G. T. Leavens, and B. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[4] K. B. Bruce, A. Fiech, and L. Petersen. Subtyping is not a good "match" for object-oriented languages. In *ECOOP*, number 1241 in LNCS, pages 104–127. Springer-Verlag, 1997.

[5] L. Cardelli and P. Wegner. On understanding types, data abstraction and polymorphism. *ACMCS*, 17(4):471–522, Dec. 1985.

[6] K. Fisher, F. Honsell, and J. C. Mitchell. A lambda calculus of objects and method specialization. *Nordic Journal of Computing*, 1(1):3–37, Spring 1994.

[7] K. Fisher and J. Mitchell. A delegation-based object calculus with subtyping. In *Fundamentals of Computation Theory (FCT'95)*, number 965 in LNCS, pages 42–61. Springer-Verlag, 1995.

[8] K. Fisher and J. Reppy. Foundations for Moby classes. Bell Labs Technical Memorandum, December 1998.

[9] K. Fisher and J. Reppy. The design of a class mechanism for Moby. In A. Press, editor, *Conference on Programming Language Design and Implementation*, pages 37–49, 1999.

[10] K. Fisher and J. Reppy. Extending Moby with inheritance-based subtyping. In *ECOOP*, number 1850 in LNCS, pages 83–107. Springer-Verlag, 2000.

[11] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183, San Diego, California, 19–21 Jan. 1998.

[12] G. Ghelli and D. Palmerini. Foundations for extensible objects with roles. Presented at the FOOL'6 workshop, Jan. 1999.

[13] P. D. Gianantonio, F. Honsell, and L. Liquori. A lambda calculus of objects with self-inflicted extension. In A. S. Notices, editor, *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, pages 166–178, October 1998.

[14] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40, Sept. 1992.

[15] B. C. Pierce and D. N. Turner. Statically typed friendly functions via partially abstract types. Technical Report ECS-LFCS-93-256, University of Edinburgh, LFCS, Apr. 1993. Also available as INRIA-Rocquencourt Rapport de Recherche No. 1899.

[16] D. Rémy. Type inference for records in a natural extension of ML. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1994.

[17] D. Rémy and J. Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Object Systems*, 4(1):27–50, 1998. A preliminary version appeared in the proceedings of the 24th ACM Conference on Principles of Programming Languages, 1997.

[18] J. G. Riecke and C. A. Stone. Privacy via subsumption. *Information and Computation*, 2000. To appear.

[19] M. Wand. Complete type inference for simple objects. In D. Gries, editor, *Second Symposium on Logic In Computer Science*, pages 207–276, Ithaca, New York, June 1987. IEEE Computer Society Press.

[20] A. K. Wright and M. Felleisen. A syntactic approach to type soundness. Technical Report COMP TR91-160, Department of Computer Science, Rice University, Houston, Texas, Apr. 1991.