An Extended Theory of Primitive Objects: First Order System

Luigi Liquori *

Dip. Informatica, Università di Torino, C.so Svizzera 185, I-10149 Torino, Italy
e-mail: liquori@di.unito.it

Abstract. We investigate a first-order extension of the Theory of Primitive Objects of [5] that supports *method extension* in presence of *object subsumption*. Extension is the ability of modifying the behavior of an object by adding new methods (and inheriting the existing ones). Object subsumption allows to use objects with a bigger interface in a context expecting another object with a smaller interface. This extended calculus has a sound type system which allows static detection of run-time errors such as *message-not-understood*, "width" subtyping and a typed equational theory on objects. Moreover, it can express classes and class-inheritance.
Categories: Type systems, design and semantics of object-oriented languages.

1 Introduction

The Abadi and Cardelli's Theory of Primitive Objects [3, 4, 5], supports method override, self-types, and (*self-type covariant*) "width" subtyping. No object extension is provided, since the objects have fixed size. In fact, the only operations allowed on objects are method invocation and method override. The objects are very simple, with just four syntactic forms, and without functions. The expressivity of the calculus is given via an encoding of the λ-calculus. The various fragments of this calculus have a sound type system that catches run-time errors such as *message-not-understood*, and a typed equational theory on objects.

The starting point of this paper is the first-order type system for the primitive object calculus, called $\mathcal{O}bj1_{\prec:}$ [5]. We extend this calculus by allowing the dynamic addition and subsumption of methods, and we provide for a sound static type system and a typed equational theory on objects. We call this (conservative) extension $\mathcal{O}bj1^{+}_{\prec:}$. The $\mathcal{O}bj1^{+}_{\prec:}$ calculus allows a considerable number of programs to be typed that are not typable in $\mathcal{O}bj1_{\prec:}$.

In this calculus, we distinguish between two "kinds" of objects-types, namely the *saturated* object-types, and the *diamond* object-types: if an object can be typed by a saturated object-type, then it can receive messages and override the methods that it contains. If an object can be typed by a diamond object-type, then it can receive messages, override some methods, and it can be extended

* Kindly supported by CSELT, Centro Studi e Laboratori Telecomunicazioni.

by new methods. On both types, a "width" subtyping relation is defined. This relation behaves differently depending on the shape of the object-type.

Summarizing, our calculus exhibits the following features:

- extendible objects with appropriate method specialization of inherited methods,
- static detection of run-time errors, such as *message-not-understood*,
- a "width" subtyping relation compatible with method extension,
- it can express classes and class-inheritance.

Moreover, the $\mathcal{O}bj1^{+}_{\prec:}$ type system can be extended with self-types by modeling the inheritance and the self-application semantics via *bounded universal polymorphism*. This (conservative w.r.t. $\mathcal{O}bj1^{+}_{\prec:}$) extension can be easily obtained with a very little cost with respect to the typing rules of $\mathcal{O}bj1^{+}_{\prec:}$ (see [16]).

This paper is organized as follows: in Section 2 we recall the untyped calculus of primitive objects, and we define the new calculus with method extension, its operational semantics and untyped equational theory. In Section 3 we present the first-order extension $\mathcal{O}bj1^{+}_{\prec:}$ with its typing and subtyping system. Section 4 is concerned with the type soundness and the typed equational theory on objects. A number of examples to give some intuition of the power of the extension are given in Section 5. Section 6 considers an interesting encoding of classes as objects that share a lot of similarities with the object-oriented language *Smalltalk-80* [15]. The last section is devoted to the comparison with the Lambda Calculus of Objects of [12] (and related papers [14, 10, 21, 9, 18, 8, 7]), Baby-Modula-3 of [1], and contains also open problems and the conclusions.

We assume that the reader is familiar with some object-oriented concepts such as delegation-based object calculi, type and subtype systems, self-types. Some knowledge of the seminal papers [12, 5] (and the above cited related papers) would be useful but not essential.

2 The Extended Primitive Calculus of Objects

2.1 The Abadi-Cardelli's Primitive Calculus

The untyped syntax of the Primitive Calculus of Objects is defined as follows:

$$\mathsf{o} ::= s \mid [\mathtt{m}_i = \varsigma(s_i)\mathsf{o}_i]^{i\in I} \mid \mathsf{o}.\mathtt{m} \mid \mathsf{o}.\mathtt{m} \prec\!\!- \varsigma(s)\mathsf{o}',$$

where in the term $[\mathtt{m}_i = \varsigma(s_i)\mathsf{o}_i]^{i\in I}$, $\mathtt{m}_i$ $(i \in I)$ are method names, o_i $(i \in I)$ are the bodies of methods, s_i $(i \in I)$ are bound parameters referring to the object itself, and ς is a binder for the s_i. Hence, an object is an unordered collection of pairs of method-names and method-bodies. If o reduces to $[\mathtt{m}_i = \varsigma(s_i)\mathsf{o}_i]^{i\in I}$, then the expression $\mathsf{o}.\mathtt{m}_i$ $(i \in I)$ stands for method invocation, and the expression $\mathsf{o}.\mathtt{m}_i \prec\!\!- \varsigma(s)\mathsf{o}'$ $(i \in I)$ stands for method override. Let $\mathsf{o} \triangleq [\mathtt{m}_i = \varsigma(s_i)\mathsf{o}_i]^{i\in I}$, and let $\mathsf{o}\{s \leftarrow \mathsf{o}'\}$ denote the substitution of the object o' for the free occurrences of s in o, and let, for $i, j \in I$, $\mathtt{m}_i$ and $\mathtt{m}_j$ be distinct methods. The operational semantics is defined as the reflexive, transitive and contextual closure of the reduction relation defined in Figure 1.

(*Select*)	$\mathtt{o.m}_j$	$\stackrel{ev}{\to}$	$\mathtt{o}_j\{s_j \leftarrow \mathtt{o}\}$	$(j \in I)$
(*Override*)	$\mathtt{o.m}_j \prec\!\!- \varsigma(s_j)\mathtt{o}'$	$\stackrel{ev}{\to}$	$[\mathtt{m}_i = \varsigma(s_i)\mathtt{o}_i, \mathtt{m}_j = \varsigma(s_j)\mathtt{o}']^{i \in I-\{j\}}$	$(j \in I)$

Fig. 1. Operational Semantics for the Primitive Calculus

(*Select*)	$\mathtt{o.m}_j$	$\stackrel{ev}{\to}$	$\mathtt{o}_j\{s_j \leftarrow \mathtt{o}\}$	$(j \in I)$
(*Override*)	$\mathtt{o} \prec\!\!- \mathtt{m}_j = \varsigma(s_j)\mathtt{o}'$	$\stackrel{ev}{\to}$	$[\mathtt{m}_i = \varsigma(s_i)\mathtt{o}_i, \mathtt{m}_j = \varsigma(s_j)\mathtt{o}']^{i \in I-\{j\}}$	$(j \in I)$
(*Extend*)	$\mathtt{o} \prec\!\!- \mathtt{m}_j = \varsigma(s_j)\mathtt{o}'$	$\stackrel{ev}{\to}$	$[\mathtt{m}_i = \varsigma(s_i)\mathtt{o}_i, \mathtt{m}_j = \varsigma(s_j)\mathtt{o}']^{i \in I}$	$(j \notin I)$

Fig. 2. Operational Semantics for the Extended Primitive Calculus

2.2 The Extended Abadi-Cardelli's Primitive Calculus

The Extended Calculus of Primitive Objects agrees with the following untyped syntax (which slightly differs from the one shown before):

$$\mathtt{o} ::= s \mid [\mathtt{m}_i = \varsigma(s_i)\mathtt{o}_i]^{i \in I} \mid \mathtt{o.m} \mid \mathtt{o} \prec\!\!- \mathtt{m} = \varsigma(s)\mathtt{o}.$$

Here the $\prec\!\!-$ operator can be intended as an override or an extension operator according to whether the method $\mathtt{m}$ belongs to the object $\mathtt{o}$ or not. The semantics of the override and of the extension is functional: an override and an extension always produce another object where the overridden method has been replaced by the new body. Therefore, the operational semantics can be given as the reflexive, transitive and contextual closure of the reduction relation defined in Figure 2.

To send the message $\mathtt{m}$ to the object $\mathtt{o}$ means to substitute the object itself (i.e. $\mathtt{o}$) in the body of $\mathtt{m}$. As usual, we do not make error conditions explicit. We can derive an untyped equational theory from the reduction rules, by simply adding rules for symmetry, transitivity, and congruence, as shown in Figure 3.

Let $\stackrel{ev}{\twoheadrightarrow}$ be the general many-step reduction. The connection between equality $\stackrel{ev}{=}$ and reduction $\stackrel{ev}{\twoheadrightarrow}$ is given by the fact that the $\stackrel{ev}{\twoheadrightarrow}$ reduction rule satisfies the Church-Rosser property.

Theorem 1 (Church-Rosser).
The relation $\stackrel{ev}{\twoheadrightarrow}$ is Church-Rosser, and if $\vdash \mathtt{o} \stackrel{ev}{=} \mathtt{o}'$, then there exists $\mathtt{o}''$ such that $\mathtt{o} \stackrel{ev}{\twoheadrightarrow} \mathtt{o}''$ and $\mathtt{o}' \stackrel{ev}{\twoheadrightarrow} \mathtt{o}''$.

Proof. The proof is standard, following the method of Tait and Martin-Löf [6].

2.3 Evaluation Strategy

In this section, we define an evaluation strategy which is directly derived from that one defined in [5]. As usual the purpose of the reduction is to maps every

$$\frac{\vdash \mathsf{o}_2 \stackrel{ev}{=} \mathsf{o}_1}{\vdash \mathsf{o}_1 \stackrel{ev}{=} \mathsf{o}_2}\ (Eq{-}Symm) \qquad \frac{\vdash \mathsf{o}_1 \stackrel{ev}{=} \mathsf{o}_2 \qquad \vdash \mathsf{o}_2 \stackrel{ev}{=} \mathsf{o}_3}{\vdash \mathsf{o}_1 \stackrel{ev}{=} \mathsf{o}_3}\ (Eq{-}Trans)$$

$$\frac{}{\vdash s \stackrel{ev}{=} s}\ (Eq{-}Var) \qquad \frac{\vdash \mathsf{o}_i \stackrel{ev}{=} \mathsf{o}'_i \qquad \forall\, i \in I}{\vdash [\mathtt{m}_i = \varsigma(s_i)\mathsf{o}_i]^{i \in I} \stackrel{ev}{=} [\mathtt{m}_i = \varsigma(s_i)\mathsf{o}'_i]^{i \in I}}\ (Eq{-}Obj)$$

$$\frac{\vdash \mathsf{o}_1 \stackrel{ev}{=} \mathsf{o}_2}{\vdash \mathsf{o}_1.\mathtt{m} \stackrel{ev}{=} \mathsf{o}_2.\mathtt{m}}\ (Eq{-}Select) \qquad \frac{\vdash \mathsf{o}_1 \stackrel{ev}{=} \mathsf{o}_2 \qquad \vdash \mathsf{o}' \stackrel{ev}{=} \mathsf{o}''}{\vdash \mathsf{o}_1 \longleftarrow\!\!\!\prec \mathtt{m} = \varsigma(s)\mathsf{o}' \stackrel{ev}{=} \mathsf{o}_2 \longleftarrow\!\!\!\prec \mathtt{m} = \varsigma(s)\mathsf{o}''}\ (Eq{-}Ext)$$

$$(\text{Let } \mathsf{o} \triangleq [\mathtt{m}_i = \varsigma(s_i)\mathsf{o}_i]^{i \in I}) \qquad \frac{j \in I}{\vdash \mathsf{o}.\mathtt{m}_j \stackrel{ev}{=} \mathsf{o}_j\{s_j \leftarrow \mathsf{o}\}}\ (Eq{-}Select_{ev})$$

$$\frac{j \in I}{\vdash \mathsf{o} \longleftarrow\!\!\!\prec \mathtt{m}_j = \varsigma(s_j)\mathsf{o}' \stackrel{ev}{=} [\mathtt{m}_i = \varsigma(s_i)\mathsf{o}_i, \mathtt{m}_j = \varsigma(s_j)\mathsf{o}']^{i \in I - \{j\}}}\ (Eq{-}Over_{ev})$$

$$\frac{j \notin I}{\vdash \mathsf{o} \longleftarrow\!\!\!\prec \mathtt{m}_j = \varsigma(s_j)\mathsf{o}' \stackrel{ev}{=} [\mathtt{m}_i = \varsigma(s_i)\mathsf{o}_i, \mathtt{m}_j = \varsigma(s_j)\mathsf{o}']^{i \in I}}\ (Eq{-}Ext_{ev})$$

Fig. 3. Untyped Equational Theory for the Extended Primitive Calculus

closed expression into a normal form, i.e. an irreducible term (if we consider constants such as natural numbers we would naturally include them among the results). We define the set of results as follows:

$$v \ ::= \ [\mathtt{m}_i = \varsigma(s_i)\mathsf{o}_i]^{i \in I} \mid wrong.$$

The result *wrong* denotes a run-time error which occurs when we send a message to an object which does not have any corresponding method, and therefore cannot respond to the message in question. The evaluation strategy *Outcome* is defined via a natural proof deduction system à la Plotkin [20] style and it is shown in Figure 4. The relation between $\stackrel{ev}{\twoheadrightarrow}$, $\stackrel{ev}{=}$ and *Outcome* is:

Proposition 2 (Soundness of *Outcome*).
If $Outcome(\mathsf{o}) = v$, *and* $v \not\equiv wrong$, *then* $\mathsf{o} \stackrel{ev}{\twoheadrightarrow} v$, *and* $\vdash \mathsf{o} \stackrel{ev}{=} v$.

Proof. By induction on the structure of the derivation of $Outcome(\mathsf{o})$.

In Section 4 we will study the relations between the *Outcome* evaluation strategy and the objects typing, by showing the "Type Soundness", i.e. that every "well typed" program will not evaluate to the *wrong* result.

3 Types

The type system of the original Primitive Calculus of Objects is composed by several fragments, each necessary to give a correct type to different objects of

$$\frac{}{Outcome([\mathtt{m}_i = \varsigma(s_i)\mathtt{o}_i]^{i \in I}) = [\mathtt{m}_i = \varsigma(s_i)\mathtt{o}_i]^{i \in I}} \quad (Red\!-\!Obj)$$

$$\frac{\begin{array}{l} Outcome(\mathtt{o}) = [\mathtt{m}_i = \varsigma(s_i)\mathtt{o}_i]^{i \in I} \\ Outcome(\mathtt{o}_j\{s_j \leftarrow [\mathtt{m}_i = \varsigma(s_i)\mathtt{o}_i]^{i \in I}\}) = v \quad j \in I \end{array}}{Outcome(\mathtt{o}.\mathtt{m}_j) = v} \quad (Red\!-\!Sel)$$

$$\frac{Outcome(\mathtt{o}) = [\mathtt{m}_i = \varsigma(s_i)\mathtt{o}_i]^{i \in I} \quad j \notin I}{Outcome(\mathtt{o} \leftarrow\!\!\!+ \mathtt{m}_j = \varsigma(s_j)\mathtt{o}') = [\mathtt{m}_i = \varsigma(s_i)\mathtt{o}_i, \mathtt{m}_j = \varsigma(s_j)\mathtt{o}']^{i \in I}} \quad (Red\!-\!Ext)$$

$$\frac{Outcome(\mathtt{o}) = [\mathtt{m}_i = \varsigma(s_i)\mathtt{o}_i]^{i \in I} \quad j \in I}{Outcome(\mathtt{o} \leftarrow\!\!\!+ \mathtt{m}_j = \varsigma(s_j)\mathtt{o}') = [\mathtt{m}_i = \varsigma(s_i)\mathtt{o}_i, \mathtt{m}_j = \varsigma(s_j)\mathtt{o}']^{i \in I - \{j\}}} \quad (Red\!-\!Over)$$

$$\frac{Outcome(\mathtt{o}) = [\mathtt{m}_i = \varsigma(s_i)\mathtt{o}_i]^{i \in I} \quad j \notin I}{Outcome(\mathtt{o}.\mathtt{m}_j) = wrong} \quad (Red\!-\!Sel\!-\!Wrong)$$

$$\frac{\begin{array}{l} Outcome(\mathtt{o}') = wrong \\ \mathtt{o} \equiv \mathtt{o}'.\mathtt{m}_j \;\; or \;\; \mathtt{o} \equiv \mathtt{o}' \leftarrow\!\!\!+ \mathtt{m}_j = \varsigma(s_j)\mathtt{o}'' \end{array}}{Outcome(\mathtt{o}) = wrong} \quad (Red\!-\!Prop\!-\!Wrong)$$

Fig. 4. Evaluation Strategy for the Extended Primitive Calculus

this calculus. For example, to give a type to those objects which contain only methods whose results are not the object itself, a first-order fragment of the type system would suffice. On the other hand, to give a type to objects whose methods returns either *self* or an updated *self* (such as, for example, a `point` object with a `move` method), recursive-types are needed. Finally, in order to include a subsumption relation between objects, the authors extend this type system with existential-types [2]. Starting from the first-order calculus $\mathcal{O}bj1_{\prec:}$ [5], we extend its type system by allowing object-extension to be typed.

In the type system of $\mathcal{O}bj1_{\prec:}$, the object-types has the following form:

$$[\mathtt{m}_i : \sigma_i]^{i \in I},$$

where we assume that the $\mathtt{m}_i$ $(i \in I)$ be distinct and that permutations do not matter. When a method $\mathtt{m}_i$ is invoked, it produces a result having the corresponding type σ_i.

As clearly stated in [13, 4], subtyping is unsound when we allow objects to be extended. As a simple example of this problem, suppose to allow extension on objects and let

$$\mathtt{point} \;\; \triangleq \;\; [\mathtt{x} = \varsigma(s)1, \mathtt{y} = \varsigma(s)s.\mathtt{x}], \qquad (1)$$

of type

$$\vdash \mathtt{point} : [\mathtt{x} : nat, \mathtt{y} : nat].$$

By subsumption (we allow for "width" subtyping, i.e. an object with some methods can be used in every context expecting an object with less methods) we get

$$\vdash \texttt{point} : [\texttt{y} : nat],$$

and by object extension we build another object

$$\texttt{point}' \triangleq \texttt{point} \leftarrow\!\!\!\prec \texttt{x} = \varsigma(s){-}1,$$

of type

$$\vdash \texttt{point}' : [\texttt{x} : int, \texttt{y} : nat],$$

which is obviously type-unsound, since

$$\texttt{point}'.\texttt{y} \stackrel{ev}{\to} [\texttt{x} = \varsigma(s){-}1, \texttt{y} = \varsigma(s)s.\texttt{x}].\texttt{y} \stackrel{ev}{\to} [\texttt{x} = \varsigma(s){-}1, \texttt{y} = \varsigma(s)s.\texttt{x}].\texttt{x} \stackrel{ev}{\to} -1,$$

with $\vdash \texttt{point}'.\texttt{y} : nat$, but $\not\vdash -1 : nat$.

Therefore, we add in $\mathcal{O}bj1^{+}_{\prec:}$ another kind of object-type, that we call *diamond-type*, of the form:

$$[\texttt{m}_i : \sigma_i \diamond \texttt{m}_j : \sigma_j]^{i \in I}_{j \in J}.$$

The symbol $\diamond$ is used to distinguish the two parts of that object-type: the *interface-part* and the *subsumption-part*. The interface-part of a diamond-type describes all the methods (and their types) that may be invoked on the objects. The subsumption-part, instead, conveys information about (the types of) methods that are (or can be) subsumed in the type-checking phase. In fact, the subsumption-part lists (a superset of) the methods (and associated types) that can be "hidden" in the object.

Intuitively, a diamond-type $[\texttt{m}_i : \sigma_i \diamond \texttt{m}_j : \sigma_j]^{i \in I}_{j \in J}$ can be assigned to an object $\texttt{o}$ with $\texttt{m}_i$ and (some of the) $\texttt{m}_j$ methods, and $\texttt{o}$ responds only to the methods listed in the interface-part. Moreover, the object can be also extended with the $\texttt{m}_j$ methods (of type σ_j) listed in the subsumption-part. At this regard, we observe that the addition of any "fresh" (i.e. unused) method $\texttt{m}$ of type σ is constrained to the prior introduction of $\texttt{m} : \sigma$ in the subsumption-part of the diamond-type via an application of a subtyping rule (see Subsection 3.2).

Accordingly, the ordinary object-types $[\texttt{m}_i : \sigma_i]^{i \in I}$ can be assigned to an object $\texttt{o}$ which responds to $\texttt{m}_i$ methods, and can be used in any context which does not extend the object $\texttt{o}$, but can override some methods, or send messages to them. In this way, ordinary object-types, here also called *saturated* object-types, can be assigned to objects which cannot be extended at all. Thus, we can distinguish two kinds of objects, with related object-types:

- objects which can be extended and overridden (typed by diamond-types of the shape $[\texttt{m}_i : \sigma_i \diamond \texttt{m}_j : \sigma_j]^{i \in I}_{j \in J}$);
- objects which can be only overridden (typed by saturated object-types of the shape $[\texttt{m}_i : \sigma_i]^{i \in I}$).

Diamond-types allow to eliminate the unsoundness previously shown. In this calculus, the subsumption rule can "hides" a method by moving it from the interface-part to its subsumption-part, and a method $\mathtt{m}$ of type σ can be added to an object $\mathtt{o}$ only if $\mathtt{m} : \sigma$ is contained in the subsumption-part of the diamond-type assigned to $\mathtt{o}$. The usual type-inclusion between (extendible) points and colored points does not hold with diamond-types, i.e.

$$[\mathtt{x} : int, \mathtt{col} : colors\ \diamond] \not<: [\mathtt{x} : int\ \diamond]$$

(and this is sound because it is well known that subsumption is not allowed in presence of object-extension), but it holds instead (see the subtyping rules)

$$\begin{aligned} [\mathtt{x} : int, \mathtt{col} : colors] &<: [\mathtt{x} : int] \\ [\mathtt{x} : int, \mathtt{col} : colors\ \diamond] &<: [\mathtt{x} : int \diamond \mathtt{col} : colors] \\ [\mathtt{x} : int\ \diamond] &<: [\mathtt{x} : int \diamond \mathtt{col} : colors] \\ [\mathtt{x} : int\ \diamond] &<: [\mathtt{x} : int]. \end{aligned}$$

The first type-inclusion gives us the desired property of using a (non extendible) colored point in any context expecting a (non extendible) ordinary point, whereas the second one is necessary for method hiding in presence of object-extension. The third inclusion ensure that an object can be extended with a new (unused) method. The last inclusion says that a diamond-type with empty subsumption-part can be also considered as a saturated object-type (this property will be generalized in the subsumption rules also to diamond-types with non empty subsumption-part).

For instance, if we take the context $[\langle\ \rangle \leftarrow\!\!\prec \mathtt{col} = \varsigma(s)red]$, then the "hole" $\langle\ \rangle$ can be filled both by a (color extensible) point object (in this case $\leftarrow\!\!\prec$ denotes method extension) and by a colored point object (where $\leftarrow\!\!\prec$ denotes method override).

As such, we can derive for the above (extendible) object $\mathtt{point}$ (1)

$$\vdash \mathtt{point} : [\mathtt{x} : nat, \mathtt{y} : nat\ \diamond],$$

and hence, by subsumption, $\vdash \mathtt{point} : [\mathtt{y} : nat \diamond \mathtt{x} : nat]$, but we cannot derive $\vdash \mathtt{point} \leftarrow\!\!\prec \mathtt{x} = \varsigma(s) - 1 : [\mathtt{x} : int, \mathtt{y} : nat\ \diamond]$, since the typing of the method $\mathtt{x}$ does not satisfy the typing inside the subsumption-part of the diamond-type in question.

3.1 Types, Contexts, and Judgments

The set of types in $\mathcal{O}bj1^{+}_{\prec:}$ is defined by the following grammar:

$$\sigma, \tau \ ::= \ \omega \mid [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]^{i \in I}_{j \in J} \mid [\mathtt{m}_i : \sigma_i]^{i \in I}.$$

We omit how to encode basic data-types and function-types, which can be treated as in [5][1]. The type-constant ω is the supertype of every type.

[1] An arrow-types $\sigma \rightarrow \tau$ is codified in the object-type $[\mathtt{arg} : \sigma, \mathtt{val} : \tau]$.

We require that, in the diamond-type $[\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]^{i \in I}_{j \in J}$, the $\mathtt{m}_i$ with $i \in I$ (resp. $\mathtt{m}_j$ with $j \in J$) are distinct, and the interface- and the subsumption-parts be *disjoint* ($I \cap J = \emptyset$), i.e. methods occurring in the former part are not occurring in the latter and vice-versa. The judgments have the following forms:

$$\Gamma \vdash ok, \qquad \Gamma \vdash \sigma, \qquad \Gamma \vdash \mathsf{o} : \sigma, \qquad \Gamma \vdash \sigma <: \tau,$$

where Γ is a context which gives types to the free variables of o, generated by the following grammar:

$$\Gamma \ ::= \ \varepsilon \mid \Gamma, s : \sigma.$$

By deriving the first two judgments we check the well-formation of the context Γ and of the type σ, respectively, while with the third one we assign a type σ to the expression o. The last judgment is the usual subtyping judgment between types. We also decorate the language with types as follows:

$$\mathsf{o} ::= s \mid [\mathtt{m}_i = \varsigma(s_i{:}\sigma_i)\mathsf{o}_i]^{i \in I} \mid \mathsf{o}.\mathtt{m} \mid \mathsf{o} \leftarrow\!\!\!\!\prec \mathtt{m} = \varsigma(s{:}\sigma)\mathsf{o}.$$

3.2 Subtyping

The subtyping relation allows to use an object of type σ in any context expecting an object of type τ, provided that $\sigma <: \tau$. The subsumption rule for objects

$$\frac{\Gamma \vdash \mathsf{o} : \sigma \qquad \Gamma \vdash \sigma <: \tau}{\Gamma \vdash \mathsf{o} : \tau} \ (<:)$$

allows an object with more methods to be used in every place where an object with less methods is required. The most important subtyping rules are presented in Figure 5 (see Appendix for the full set of rules).

The ($Shift_\diamond$) rule says that we can "hide" a method which belongs to the interface-part simply by moving it into the subsumption-part of the diamond-type. This rule is needed when we know that the hidden method will be added again. This subtyping rule allows to use subsumption over extendible objects.

The ($Extend_\diamond$) rule says that an object with smaller subsumption-part can be used in any context which expects an object with a bigger subsumption-part. This rule is crucial to ensure that an object can be dynamically extended with fresh methods.

The ($Sat_\diamond$) rule says that a diamond-type becomes a saturated object-type preserving only the methods in the interface-part. When this rule is applied, the "extendible" object to which is assigned a saturated object-type becomes a "non-extendible" one.

The ($Width$) rule hides a method from the interface-part of the saturated object-type in question. Note that, when a method is hidden by using this rule, the hidden method cannot be recovered. We stress, again, that when a saturated object-type is assigned to an object, that object cannot be extended, but it can be used, subsumed and overridden. This subtyping rule correspond to the ordinary subtyping for objects of [5].

$$\frac{\Gamma \vdash [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]^{i \in I+K}_{j \in J}}{\Gamma \vdash [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]^{i \in I+K}_{j \in J} <: [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]^{i \in I}_{j \in J+K}} \quad (Shift_\diamond)$$

$$\frac{\Gamma \vdash [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]^{i \in I}_{j \in J+K}}{\Gamma \vdash [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]^{i \in I}_{j \in J} <: [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]^{i \in I}_{j \in J+K}} \quad (Extend_\diamond)$$

$$\frac{\Gamma \vdash [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]^{i \in I}_{j \in J}}{\Gamma \vdash [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]^{i \in I}_{j \in J} <: [\mathtt{m}_i : \sigma_i]^{i \in I}} \quad (Sat_\diamond)$$

$$\frac{\Gamma \vdash [\mathtt{m}_i : \sigma_i]^{i \in I+J}}{\Gamma \vdash [\mathtt{m}_i : \sigma_i]^{i \in I+J} <: [\mathtt{m}_i : \sigma_i]^{i \in I}} \quad (Width)$$

Fig. 5. Main Subtyping Rules for $\mathcal{O}bj1^{+}_{\prec:}$

3.3 Typing

The main typing rules are shown in Figure 6 where we assume that when $i, j \in I$, $\mathtt{m}_i \neq \mathtt{m}_j$ (see Appendix for the full set of rules). In addition to these rules, we have also rules for well-formation of contexts. By inspecting the typing rules, one can see that the $(Object)$ rule is the same as in $\mathcal{O}bj1_{\prec:}$ when we let $H_i \stackrel{set}{=} I$, and $[\mathtt{m}_i : \sigma_i \diamond]^{i \in I} \triangleq [\mathtt{m}_i : \sigma_i]^{i \in I}$ (in fact, in the original calculus, we can build only "fixed-size" objects). Also the $(Over)$ rule is the same as in $\mathcal{O}bj1_{\prec:}$ if we assume $\tau \equiv [\mathtt{m}_i : \sigma_i]^{i \in I}$. As such, $\mathcal{O}bj1^{+}_{\prec:}$ is a proper extension of $\mathcal{O}bj1_{\prec:}$. We only explain the (Ext) rule, the new one; firstly, one can see that we cannot extend an object whose object-type is saturated. Secondly, this rule allows one to extend an object with a new method if and only if that method is present in the subsumption-part of the diamond-type assigned to the object to be extended. But this condition can always be satisfied by a diamond-type thanks to the subtyping rule $(Extend_\diamond)$. Finally, observe that this rule handles also the case where the method belongs to $\mathtt{o}$ but it has been already subsumed.

An important difference with [5] is that here the ς-bound variables s_i (referring to *self*) in the same object $\mathtt{o}$ can have different saturated object-types. This fits well with the semantics of the message send thanks to the presence of the subsumption rule $(<:)$.

3.4 Typing à la Curry

We could also omit type-decorations inside ς-binders and build a "type inference" version of $\mathcal{O}bj1^{+}_{\prec:}$, by adopting the untyped calculus instead of the explicitly typed one. The operational semantics, the typing and subtyping rules are the same as in Figure 2 and 5, and 6 (taking into account the modification in the

$$\frac{\begin{array}{l}\Gamma \vdash \sigma_i \quad \forall\, i \in I\\ \Gamma \vdash \sigma_j \quad \forall j \in J \quad I \cap J = \emptyset\end{array}}{\Gamma \vdash [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]^{i \in I}_{j \in J}}\;(\mathit{Diamond{-}Type}) \qquad \frac{\Gamma \vdash \sigma_i \quad \forall\, i \in I}{\Gamma \vdash [\mathtt{m}_i : \sigma_i]^{\,i \in I}}\;(\mathit{Sat{-}Type})$$

(Let $\tau_i \triangleq [\mathtt{m}_h : \sigma_h]^{h \in H_i}$).

$$\frac{\Gamma, s_i : \tau_i \vdash \mathsf{o}_i : \sigma_i \qquad \forall i \in I \qquad H_i \subseteq I}{\Gamma \vdash [\mathtt{m}_i = \varsigma(s_i{:}\tau_i)\mathsf{o}_i]^{\,i \in I} : [\mathtt{m}_i : \sigma_i \;\diamond]^{\,i \in I}}\;(\mathit{Object}) \qquad \frac{\Gamma \vdash \mathsf{o} : [\mathtt{m}_k : \sigma_k]}{\Gamma \vdash \mathsf{o}.\mathtt{m}_k : \sigma_k}\;(\mathit{Select})$$

$$\frac{\Gamma \vdash \mathsf{o} : \tau \qquad \Gamma \vdash \tau {<:} [\mathtt{m}_i : \sigma_i]^{\,i \in I} \qquad \Gamma, s_k : [\mathtt{m}_i : \sigma_i]^{\,i \in I} \vdash \mathsf{o}' : \sigma_k \qquad k \in I}{\Gamma \vdash \mathsf{o} \leftarrow\!\!\!+ \mathtt{m}_k = \varsigma(s_k{:}[\mathtt{m}_i : \sigma_i]^{\,i \in I})\mathsf{o}' : \tau}\;(\mathit{Over})$$

(Let $\tau_k \triangleq [\mathtt{m}_h : \sigma_h]^{h \in H}$).

$$\frac{\Gamma \vdash \mathsf{o} : [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]^{i \in I}_{j \in J} \qquad \Gamma, s_k : \tau_k \vdash \mathsf{o}' : \sigma_k \qquad H \subseteq I \qquad k \in J}{\Gamma \vdash \mathsf{o} \leftarrow\!\!\!+ \mathtt{m}_k = \varsigma(s_k{:}\tau_k)\mathsf{o}' : [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]^{i \in I + \{k\}}_{j \in J - \{k\}}}\;(\mathit{Ext})$$

Fig. 6. Main Typing Rules for $\mathcal{O}bj1^{+}_{\prec:}$

syntax). Type inference for primitive objects has been extensively studied in [19] (see also Subsection 7.1).

4 Soundness and Equational Theory of $\mathcal{O}bj1^{+}_{\prec:}$

In this section, we prove that the $\mathcal{O}bj1^{+}_{\prec:}$ type system is sound. Because of lack of space, all proof are omitted (the reader is referred to [16] for detailed proofs).

The following fact is crucial for subject reduction.

Fact 3 (Sub Methods).

1. *If* $\Gamma \vdash [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]^{i \in I}_{j \in J} {<:} [\mathtt{m}_h : \sigma_h \diamond \mathtt{m}_k : \sigma_k]^{h \in H}_{k \in K}$, *then* $H \subseteq I$, *and* $J \subseteq K$, *and* $I \subseteq H \cup K$.
2. *If* $\Gamma \vdash [\mathtt{m}_i = \varsigma(s_i{:}\tau_i)\mathsf{o}_i]^{\,i \in I} : [\mathtt{m}_h : \sigma_h \diamond \mathtt{m}_k : \sigma_k]^{h \in H}_{k \in K}$, *then* $H \subseteq I$, *and* $I - H \subseteq K$.

The following two lemmas are useful for stating structural properties on objects and to guarantee that the calculus is closed under substitution.

Lemma 4 (Generation).

1. *(Bodies) If* $\Gamma \vdash [\mathtt{m}_i = \varsigma(s_i{:}\tau_i)\mathsf{o}_i]^{\,i \in I} : \tau$, *then there exists* $\{\sigma_i \mid i \in I\}$, *such that* $\Gamma, s_i : \tau_i \vdash \mathsf{o}_i : \sigma_i$, *and* $[\mathtt{m}_i : \sigma_i \;\diamond]^{\,i \in I} {<:} \tau$.
2. *(Object) If* $\Gamma \vdash [\mathtt{m}_i = \varsigma(s_i{:}\tau_i)\mathsf{o}_i]^{\,i \in I} : \tau$, *then there exists* $\{\sigma_i \mid i \in I\}$, *such that* $\Gamma \vdash [\mathtt{m}_i = \varsigma(s_i{:}\tau_i)\mathsf{o}_i]^{\,i \in I} : [\mathtt{m}_i : \sigma_i \;\diamond]^{\,i \in I}$, $\Gamma \vdash [\mathtt{m}_i : \sigma_i \;\diamond]^{\,i \in I} {<:} \tau$, *and for all* $j \in I$, $\Gamma \vdash [\mathtt{m}_i : \sigma_i \;\diamond]^{\,i \in I} {<:} \tau_j$.

Proof. Both parts can be proved by induction on the structure of derivations.

Lemma 5 (Substitution).
If $\Gamma, s : \tau \vdash \mathsf{o} : \rho$ and $\Gamma \vdash \mathsf{o}' : \sigma$ and $\Gamma \vdash \sigma <: \tau$, then $\Gamma \vdash \mathsf{o}\{s \leftarrow \mathsf{o}'\} : \rho$.

Proof. By induction on the structure of derivations.

We can now prove the subject reduction theorem.

Theorem 6 (Subject Reduction for $\mathcal{O}bj1^{+}_{\prec:}$).
If $\Gamma \vdash \mathsf{o} : \sigma$ and $\mathsf{o} \overset{ev}{\to} \mathsf{o}'$, then $\Gamma \vdash \mathsf{o}' : \sigma$.

Proof. By cases on the definition of $\overset{ev}{\to}$, using Fact 3, Lemmas 4, and 5.

We can now prove the type soundness result that certifies that every well typed program cannot evaluate to the *wrong* result.

Theorem 7 (Type Soundness for $\mathcal{O}bj1^{+}_{\prec:}$).
Let o be a closed expression. If $\varepsilon \vdash \mathsf{o} : \sigma$ and $Outcome(\mathsf{o})$ is defined, then $Outcome(\mathsf{o}) \neq wrong$.

Proof. By induction on the structure of the derivation of $Outcome(\mathsf{o})$.

Moreover it holds:

Theorem 8 ($\mathcal{O}bj1^{+}_{\prec:}$ has Minimum Types).
If $\Gamma \vdash \mathsf{o} : \sigma$, then there exists σ' such that $\Gamma \vdash \mathsf{o} : \sigma'$, and for any σ'', if $\Gamma \vdash \mathsf{o} : \sigma''$, then $\Gamma \vdash \sigma' <: \sigma''$.

Proof. The proof is standard and follows the guidelines of [5].

As in the original calculus, the lack of type-annotations inside ς-binders destroy the minimum-type property for the type inference version of $\mathcal{O}bj1^{+}_{\prec:}$ [5].

4.1 An Equational Type Theory for $\mathcal{O}bj1^{+}_{\prec:}$

In this section we present the "typed" equational theory for $\mathcal{O}bj1^{+}_{\prec:}$. We refine the untyped theory presented in Section 2 by introducing a typing judgment to ensure that equal (provable in the theory) terms have the same type. We introduce the judgment:

$$\Gamma \vdash \mathsf{o} \overset{ev}{=} \mathsf{o}' : \tau,$$

to describe the property that o and o' are provably equal in the theory with type τ. Figure 7 presents the most important rules while the full set of rules can be found in the Appendix.

The relation between $\overset{ev}{=}$, $\overset{ev}{\to}$, *Outcome*, and the equational type theory is:

Theorem 9 (Soundness of Outcome w.r.t. the Equational Theory).
Let o be a closed expression. If $\varepsilon \vdash \mathsf{o} : \sigma$ and $Outcome(\mathsf{o}) = v$, then $\mathsf{o} \overset{ev}{\twoheadrightarrow} v$, and $\varepsilon \vdash \mathsf{o} \overset{ev}{=} v : \sigma$.

Proof. The proof follows from Proposition 2, and Theorems 6, and 7.

$$\text{(Let } \tau_i \triangleq [\mathtt{m}_h : \sigma_h]^{h \in H_i}\text{, and } \tau_j \triangleq [\mathtt{m}_k : \sigma_k]^{k \in K_j}\text{, and } \tau \triangleq [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]_{j \in J}^{i \in I}\text{).}$$

$$\frac{\begin{array}{lll} \Gamma, s_i : \tau_i \vdash \mathtt{o}_i : \sigma_i & \forall\, i \in I & H_i \subseteq I \\ \Gamma, s_j : \tau_j \vdash \mathtt{o}_j : \sigma_j & \forall j \in J & K_j \subseteq I{+}J \end{array}}{\Gamma \vdash [\mathtt{m}_i = \varsigma(s_i{:}\tau_i)\mathtt{o}_i]^{i \in I} \stackrel{ev}{=} [\mathtt{m}_i = \varsigma(s_i{:}\tau_i)\mathtt{o}_i, \mathtt{m}_j = \varsigma(s_j{:}\tau_j)\mathtt{o}_j,]_{j \in J}^{i \in I} : \tau} \quad (Eq{-}Obj{-}{<:})$$

(In the next rules, let
$\mathtt{o} \triangleq [\mathtt{m}_z = \varsigma(s_z{:}\tau_z)\mathtt{o}_z]^{z \in Z}$, and $\tau_k \triangleq [\mathtt{m}_h : \sigma_h]^{h \in H}$, and $\tau \triangleq [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]_{j \in J - \{k\}}^{i \in I + \{k\}}$).

$$\frac{\Gamma \vdash \mathtt{o} : [\mathtt{m}_k : \sigma_k]}{\Gamma \vdash \mathtt{o.m}_k \stackrel{ev}{=} \mathtt{o}_k\{s_k \leftarrow \mathtt{o}\} : \sigma_k} \quad (Eq{-}Select_{ev})$$

$$\frac{\Gamma \vdash \mathtt{o} : [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]_{j \in J}^{i \in I} \quad \Gamma, s_k : \tau_k \vdash \mathtt{o}' : \sigma_k \quad H \subseteq I \quad k \in Z{-}I}{\Gamma \vdash \mathtt{o} \leftarrow\!\!\!\prec \mathtt{m}_k = \varsigma(s_k{:}\tau_k)\mathtt{o}' \stackrel{ev}{=} [\mathtt{m}_z = \varsigma(s_z{:}\tau_z)\mathtt{o}_z, \mathtt{m}_k = \varsigma(s_k{:}\tau_k)\mathtt{o}']^{z \in Z - \{k\}} : \tau} \quad (Eq{-}Ext^1_{ev})$$

$$\frac{\Gamma \vdash \mathtt{o} : [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]_{j \in J}^{i \in I} \quad \Gamma, s_k : \tau_k \vdash \mathtt{o}' : \sigma_k \quad H \subseteq I \quad k \in J{-}Z}{\Gamma \vdash \mathtt{o} \leftarrow\!\!\!\prec \mathtt{m}_k = \varsigma(s_k{:}\tau_k)\mathtt{o}' \stackrel{ev}{=} [\mathtt{m}_z = \varsigma(s_z{:}\tau_z)\mathtt{o}_z, \mathtt{m}_k = \varsigma(s_k{:}\tau_k)\mathtt{o}']^{z \in Z} : \tau} \quad (Eq{-}Ext^2_{ev})$$

Fig. 7. Typed Equational Theory for $\mathcal{O}bj1^{+}_{\prec:}$

As in the original calculus, we may find two objects which may give equal result for all their methods, and still be distinguishable in the equational theory. As a simple example of such two objects, let

$$\begin{array}{lll} \mathtt{o}_1 & \triangleq & [\mathtt{x} = \varsigma(s{:}[\,])1, \mathtt{y} = \varsigma(s{:}[\,])1], \\ \mathtt{o}_2 & \triangleq & [\mathtt{x} = \varsigma(s{:}[\,])1, \mathtt{y} = \varsigma(s{:}[\mathtt{x}{:}int])s.\mathtt{x}], \end{array}$$

both of type $[\mathtt{x} : int, \mathtt{y} : int]$, but $\not\vdash \mathtt{o}_1 \stackrel{ev}{=} \mathtt{o}_2 : [\mathtt{y} : int]$. More details can be found in [4].

5 Examples

In this section, we will present a few examples that help to illustrate the features of our first order type system.

Example 1 (Method Specialization). We show how our typing rules can capture the desired form of method specialization by extending the object $[\mathtt{x} = \varsigma(s{:}[\,])1]$ with a $\mathtt{y}$ field which depends on the $\mathtt{x}$ field, so building an (extendible) diagonal point. Consider the following object expression:

$$\mathtt{point} \;\triangleq\; [\mathtt{x} = \varsigma(s{:}[\,])1] \leftarrow\!\!\!\prec \mathtt{y} = \varsigma(s{:}[\mathtt{x}{:}int])s.\mathtt{x}.$$

We can derive $\varepsilon \vdash \mathtt{point} : [\mathtt{x} : int, \mathtt{y} : int \,\diamond]$, with the following derivation:

$$
\dfrac{\dfrac{\dfrac{s:[\,]\vdash 1:int}{\varepsilon\vdash[\mathtt{x}=\varsigma(s{:}[\,])1]:[\mathtt{x}:int\ \diamond]}\ (Object)}{\varepsilon\vdash[\mathtt{x}=\varsigma(s{:}[\,])1]:[\mathtt{x}:int\diamond\mathtt{y}:int]}\ (<:)\qquad \dfrac{s:[\mathtt{x}{:}int]\vdash s:[\mathtt{x}:int]}{s:[\mathtt{x}{:}int]\vdash s.\mathtt{x}:int}\ (Select)}{\varepsilon\vdash\mathtt{point}:[\mathtt{x}:int,\mathtt{y}:int\ \diamond]}\ (Ext)
$$

Example 2 (Object-Internal Subtyping). In this example we show that our type system allows also subtyping inside objects. We introduce λ-binders to denote functions (see [5]). Let the following object-types:

$$
\begin{array}{lcll}
P_\diamond & \triangleq & [\mathtt{x}:int\ \diamond] & \text{(extendible point)}\\
P_{col,y} & \triangleq & [\mathtt{x}:int\diamond\mathtt{y}:int,\mathtt{col}:colors] & \text{(point extendible with } \mathtt{y} \text{ and } \mathtt{col})\\
2P_\diamond & \triangleq & [\mathtt{x}:int,\mathtt{y}:int\ \diamond] & \text{(extendible bidimensional point)}\\
CP & \triangleq & [\mathtt{x}:int,\mathtt{col}:colors] & \text{(non-extendible colored point).}
\end{array}
$$

For the object `foo`

$$
\begin{array}{lcl}
\mathtt{foo} & \triangleq & [\mathtt{addcol}=\varsigma(s{:}[\,])\lambda p{:}P_{col,y}.p\leftarrow\!\!\prec\mathtt{col}=\varsigma(s'{:}[\,])red,\\
 & & \ \mathtt{select}=\varsigma(s{:}[\mathtt{addcol}{:}P_{col,y}\to CP,\mathtt{get\_p}{:}P_\diamond,\mathtt{get\_2p}{:}2P_\diamond])\lambda b{:}bool.\\
 & & \qquad \text{if } b=true \text{ then } s.\mathtt{addcol}(s.\mathtt{get\_p}) \text{ else } s.\mathtt{addcol}(s.\mathtt{get\_2p}),\\
 & & \ \mathtt{get\_p}\ \ =\varsigma(s{:}[\,])[\mathtt{x}=\varsigma(s{:}[\,])1],\\
 & & \ \mathtt{get\_2p}=\varsigma(s{:}[\,])[\mathtt{x}=\varsigma(s{:}[\,])1,\mathtt{y}=\varsigma(s{:}[\,])1]\\
 & &],
\end{array}
$$

we can derive

$$\vdash\mathtt{foo}:[\mathtt{addcol}:P_{col,y}\to CP,\mathtt{select}:bool\to CP,\mathtt{get\_p}:P_\diamond,\mathtt{get\_2p}:2P_\diamond]$$

and

$$\vdash\mathtt{foo.select}\ true:CP,\qquad\qquad \vdash\mathtt{foo.select}\ false:CP.$$

This is possible since $P_\diamond <: P_{col,y}$, and $2P_\diamond <: P_{col,y}$.

Note that other typing for `foo` are possible: among the others we mention the following interesting one:

$$\vdash\mathtt{foo}:[\mathtt{addcol}:P_{col,y}\to CP_y,\mathtt{select}:bool\to CP_y,\mathtt{get\_p}:P_\diamond,\mathtt{get\_2p}:2P_\diamond],$$

where

$$CP_y\triangleq[\mathtt{x}:int,\mathtt{col}:colors\diamond\mathtt{y}:int],$$

which allows one to type the interesting programs

$$\vdash\mathtt{foo.select}\ true\leftarrow\!\!\prec\mathtt{y}=\varsigma(s)1:[\mathtt{x}:int,\mathtt{y}:int,\mathtt{col}:colors\ \diamond],$$

and

$$\vdash\mathtt{foo.select}\ false\leftarrow\!\!\prec\mathtt{y}=\varsigma(s)1:[\mathtt{x}:int,\mathtt{y}:int,\mathtt{col}:colors\ \diamond].$$

Both programs produce an extendible bidimensional colored point.

Example 3 (Object Subtyping). Let `point` as in the Example 1 and let `c_point` be obtained by extending `point` with a `col` field. By an inspection of the typing rules we derive

$$\vdash \texttt{point} : P_\diamond, \qquad \vdash \texttt{c\_point} : CP_\diamond,$$

where

$$\begin{array}{lcllcl} P & \triangleq & [\texttt{x} : int] & CP & \triangleq & [\texttt{x} : int, \texttt{col} : colors] \\ P_\diamond & \triangleq & [\texttt{x} : int \ \diamond] & CP_\diamond & \triangleq & [\texttt{x} : int, \texttt{col} : colors \ \diamond]. \end{array}$$

Now consider the following programs and related (derivable) types:

$$\begin{array}{lcll} f_1 & \triangleq & \lambda s{:}P.s.\texttt{x} & : \ P \to int \\ f_2 & \triangleq & \lambda s{:}P.s \leftarrow\!\!\!\prec \texttt{x} = \varsigma(s'{:}[\,])2 & : \ P \to P \\ f_3 & \triangleq & \lambda s{:}P_\diamond.s \leftarrow\!\!\!\prec \texttt{col} = \varsigma(s'{:}[\,])red & : \ P_\diamond \to CP_\diamond. \end{array}$$

Again, by inspecting the typing rules, we get that the following judgments are derivable:

$$\begin{array}{lll@{\qquad}lll} \vdash & f_1 \ \texttt{point} & : int & \vdash & f_1 \ \texttt{c\_point} & : int \\ \vdash & f_2 \ \texttt{point} & : P & \vdash & f_2 \ \texttt{c\_point} & : P \\ \vdash & f_3 \ \texttt{point} & : CP_\diamond & \nvdash & f_3 \ \texttt{c\_point} & : CP_\diamond. \end{array}$$

The last judgment is correctly false since $CP_\diamond \not<: P_\diamond$.

6 Classes-as-Objects

In this section, we show how the functional object calculus $\mathcal{O}bj1^{+}_{<:}$ can easily codify classes as objects; here we give a simple first-order encoding of classes, metaclasses, and instances, that share a lot of similarities with the object-oriented language *Smalltalk-80* [15]. This encoding shares both the class- and the delegation-based object-oriented styles of programming, because it allows to build classes and instances (and assign a type in presence of a "width" subtyping relation) and to extend and override dynamically some object methods. In the object-oriented jargon, creating an instance of a given class can be viewed as an activity that must be *delegated* to some object. Then the following question arises: which object should have the responsibility for this activity? One solution places a layer of management between the user, who desires the creation of a new object, and the code that performs the allocation of the memory. It follows that, for each class, say `A`, to be defined, we have a corresponding proper object, that has the responsibility of creating instances of `A`. We call that object (the *metaclass*) `A_Class`; it must have all the information about the size of the class it represents, the methods to which instances of this class will respond and a method `New`, that performs the creation of the class `A`. As such, the class `A` is an instance of the (meta)class `A_Class`. The class `A`, in turn contains a method, called `new`, that performs the creation of instances of the class `A`. Figure 8 depicts the class and subclass hierarchy.

In the next subsections, we present the encoding of the metaclasses, classes and instances of Figure 8. In particular, we present the encoding of the classes

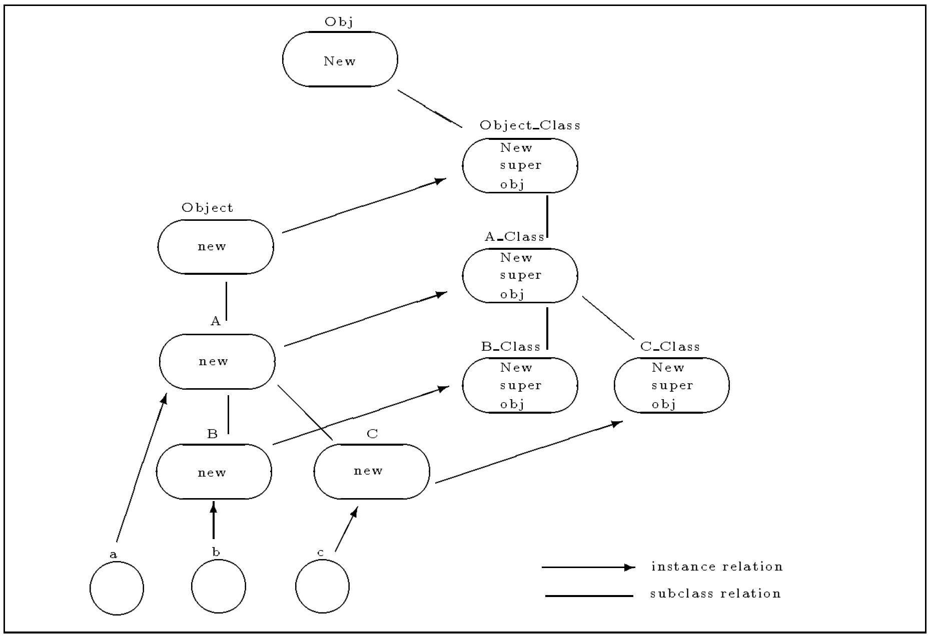

Fig. 8. The Class-Subclass Hierarchy

and metaclasses of points and colored points. Then, we present the typing of those objects in the $\mathcal{O}bj1^{+}_{\prec:}$ type system. It is worth noting that the only objects that need to be built are the objects representing metaclasses and the top level object `Obj`: objects representing classes and instances are generated via message sending and successive reductions.

6.1 Metaclasses

Let the following types:

$$\begin{array}{lllllll}
\sigma_1 & \triangleq & [\texttt{New} : [\diamond]] & \quad & \sigma_5 & \triangleq & [\texttt{New} : \sigma_2, \texttt{super} : \sigma_1, \texttt{obj} : [\diamond]] \\
\sigma_2 & \triangleq & [\texttt{new} : [\diamond]] & & \sigma_6 & \triangleq & [\texttt{New} : \sigma_3, \texttt{super} : \sigma_5, \texttt{obj} : P_\diamond] \\
\sigma_3 & \triangleq & [\texttt{new} : P_\diamond] & & \sigma_7 & \triangleq & [\texttt{New} : \sigma_4, \texttt{super} : \sigma_6, \texttt{obj} : CP_\diamond] \\
\sigma_4 & \triangleq & [\texttt{new} : CP_\diamond]. & & & &
\end{array}$$

The metaclasses `Obj_Class`, `Point_Class`, and `C_Point_Class`, together with the top-level object of the hierarchy `Obj`, are encoded as follows:

$$\begin{array}{rcl}
\texttt{Obj} & \triangleq & [\texttt{New} = \varsigma(s{:}[\,])[\,]] \\
\texttt{Obj_Class} & \triangleq & [\texttt{New} = \varsigma(s{:}[\texttt{obj}{:}[\diamond]])[\texttt{new} = \varsigma(s'{:}[\,])s.\texttt{obj}], \\
& & \;\texttt{super} = \varsigma(s{:}[\,])\texttt{Obj}, \\
& & \;\texttt{obj} = \varsigma(s{:}[\texttt{super}{:}\sigma_1])s.\texttt{super}.\texttt{New}]
\end{array}$$

$$\begin{array}{rcl}
\texttt{Point_Class} & \triangleq & [\texttt{New} = \varsigma(s{:}[\texttt{obj}{:}P_\diamond])[\texttt{new} = \varsigma(s'{:}[\,])s.\texttt{obj}], \\
& & \;\texttt{super} = \varsigma(s{:}[\,])\texttt{Obj_Class}, \\
& & \;\texttt{obj} = \varsigma(s{:}[\texttt{super}{:}\sigma_5])s.\texttt{super.obj} \leftarrow\!\!\!\!\!\prec \texttt{x} = \varsigma(s'{:}[\,])1] \\
\texttt{C_Point_Class} & \triangleq & [\texttt{New} = \varsigma(s{:}[\texttt{obj}{:}CP_\diamond])[\texttt{new} = \varsigma(s'{:}[\,])s.\texttt{obj}], \\
& & \;\texttt{super} = \varsigma(s{:}[\,])\texttt{Point_Class}, \\
& & \;\texttt{obj} = \varsigma(s{:}[\texttt{super}{:}\sigma_6])s.\texttt{super.obj} \leftarrow\!\!\!\!\!\prec \texttt{col} = \varsigma(s'{:}[\,])red].
\end{array}$$

The meaning of the methods of the above metaclasses is as follows. When the method `New` is invoked on the metaclasses[2] (i.e `Point_Class` and `C_Point_Class`), it produces as result another object, which is the class representing all the objects instances (e.g. points and colored points, respectively). The `super` method contains a pointer of the direct super(meta)class of the class to be defined. Finally, the method `obj` contains a copy of the object instance of the class; that object will be cloned when an instance of the class is created (by sending the message `new` to the class).

It is easy to verify that the above objects can be typed in $\mathcal{O}bj1^{+}_{\prec:}$ by the following derivable types:

$$\begin{array}{lllll@{\qquad}lllll}
\varepsilon & \vdash & \texttt{Obj} & : & \sigma_1 & \varepsilon & \vdash & \texttt{Point_Class} & : & \sigma_6 \\
\varepsilon & \vdash & \texttt{Obj_Class} & : & \sigma_5 & \varepsilon & \vdash & \texttt{C_Point_Class} & : & \sigma_7.
\end{array}$$

6.2 Classes

The classes `Object`, `Point`, and `C_Point`, are encoded in $\mathcal{O}bj1^{+}_{\prec:}$ as follows:

$$\begin{array}{rcl}
\texttt{Object} & \triangleq & [\texttt{new} = \varsigma(s{:}[\,])[\,]] \\
\texttt{Point} & \triangleq & [\texttt{new} = \varsigma(s{:}[\,])[\,] \leftarrow\!\!\!\!\!\prec \texttt{x} = \varsigma(s'{:}[\,])1] \\
\texttt{C_Point} & \triangleq & [\texttt{new} = \varsigma(s{:}[\,])[\,] \leftarrow\!\!\!\!\!\prec \texttt{x} = \varsigma(s'{:}[\,])1 \leftarrow\!\!\!\!\!\prec \texttt{col} = \varsigma(s'{:}[\,])red]],
\end{array}$$

with the following derivable judgments in the equational theory

$$\begin{array}{llcll}
\varepsilon \vdash & \texttt{Object} & \stackrel{ev}{=} & [\texttt{new} = \varsigma(s{:}[\,])[\,]] & : \sigma_2 \\
\varepsilon \vdash & \texttt{Point} & \stackrel{ev}{=} & [\texttt{new} = \varsigma(s{:}[\,])[\texttt{x} = \varsigma(s'{:}[\,])1]] & : \sigma_3 \\
\varepsilon \vdash & \texttt{C_Point} & \stackrel{ev}{=} & [\texttt{new} = \varsigma(s{:}[\,])[\texttt{x} = \varsigma(s'{:}[\,])1, \texttt{col} = \varsigma(s'{:}[\,])red]] & : \sigma_4,
\end{array}$$

and it holds

$$\begin{array}{rcl}
\texttt{Obj_Class.New} & \stackrel{ev}{\twoheadrightarrow} & \texttt{Object} \\
\texttt{Point_Class.New} & \stackrel{ev}{\twoheadrightarrow} & \texttt{Point} \\
\texttt{C_Point_Class.New} & \stackrel{ev}{\twoheadrightarrow} & \texttt{C_Point}.
\end{array}$$

[2] The `New` method should be "fired" automatically and only once on all defined metaclasses.

6.3 Instances

When the method `new` is invoked on the classes `Point` and `C_Point`, it produces as results points and colored points objects instances, respectively. This means that

$$\begin{aligned}
\texttt{Object.new} &\overset{ev}{\twoheadrightarrow} [\,] \\
\texttt{Point.new} &\overset{ev}{\twoheadrightarrow} [\texttt{x} = \varsigma(s'{:}[\,])1] \\
\texttt{C_Point.new} &\overset{ev}{\twoheadrightarrow} [\texttt{x} = \varsigma(s'{:}[\,])1, \texttt{col} = \varsigma(s'{:}[\,])red],
\end{aligned}$$

and

$$\begin{aligned}
\varepsilon \vdash [\,] &: [\diamond] \\
\varepsilon \vdash [\texttt{x} = \varsigma(s'{:}[\,])1] &: P_\diamond \\
\varepsilon \vdash [\texttt{x} = \varsigma(s'{:}[\,])1, \texttt{col} = \varsigma(s'{:}[\,])red] &: CP_\diamond.
\end{aligned}$$

6.4 Discussion

We have presented a functional encoding of classes and metaclasses in terms of typable objects of our Extended Calculus of Primitive Objects. The objects of this encoding are typable in the first-order system $\mathcal{O}bj1^{+}_{\prec:}$, provided that no method will return the object itself or an update of *self*. This encoding agrees with both the class- and the delegation-based object-oriented styles of programming. The soundness of the type system guarantees that every program will not go into the *message-not-understood* run-time error. The class-subclass hierarchy shares a lot of similarities with the one of *Smalltalk-80*. Although not treated here, it is not difficult to complete this encoding with *class* and *instance* methods and fields (we recall that a field is a methods that does not make use of *self*).

We point out that color points and points metaclasses *cannot* be obtained by object override from point and object metaclasses, respectively, since we do not have "depth" subtyping. For the same reason, color points and points classes *cannot* be obtained by object override from point and object classes. Moreover, we observe that color points and points instances *can* be obtained by inheritance (i.e. by successive object extensions) from the point and object instances, respectively, i.e:

$$\begin{aligned}
[\,] \leftarrow\!\!\!\prec \texttt{x} = \varsigma(s'{:}[\,])1 &\overset{ev}{\to} [\texttt{x} = \varsigma(s'{:}[\,])1] \\
[\texttt{x} = \varsigma(s'{:}[\,])1] \leftarrow\!\!\!\prec \texttt{col} = \varsigma(s'{:}[\,])red &\overset{ev}{\to} [\texttt{x} = \varsigma(s'{:}[\,])1, \texttt{col} = \varsigma(s'{:}[\,])red],
\end{aligned}$$

even if $CP_\diamond \not\prec: P_\diamond \not\prec: [\diamond]$, thanks to our subtyping rules. Finally, note that this encoding will assign diamond-types to class-instances; as such, class instances can be extended and overridden in pure delegation object-oriented style.

The above encoding is not the only possible one; as an example, if we want to turn into a simple class-based style of programming, then we can drop the possibility of dynamically extending and overriding class instances. Then, we could design a type system for $\mathcal{O}bj1^{+}_{\prec:}$, where we can distinguish between two kinds of objects:

- objects (contained inside the methods of the classes and metaclasses) which can be extended and overridden (but not subsumed);
- objects (representing class instances) which can be only used via message sending, with saturated object-types and whose object-types agrees with a "depth-width" subtyping relation.

The resulting type system will have, although not in details but in the spirit, some similarities with [14] (see the related work). Our experience says that the above presented mixed "class-delegation" style of programming is more flexible and powerful than the "class-only" one. The following simple example show how classes, objects, and object extensions can be easily integrated thanks to our subtyping relation among object-types.

Example 4. Let the `Point` and `C_Point` classes be defined as in Subsection 6.2, and consider the following program (let $P_{col} \triangleq [\mathtt{x} : int \diamond \mathtt{col} : colors]$):

$$f \triangleq \lambda s{:}P_{col}.(s \longleftarrow \mathtt{col}{:}\varsigma(s{:}[\,])red).\mathtt{x} : P_{col} \rightarrow int.$$

This function will accept as input both an instance of the `Point` and `C_Point` classes, and as a consequence the following judgments are derivable: f `Point.new` : int, and f `C_Point.new` : int.

7 Related Work

Among the many object-based languages we find in the literature, we recall the following ones.

M.Abadi, in [1], presents a small functional language which include the main features of Modula-3. This language allows object override, a small form of object extension and "width" subtyping. The soundness of the typing system is guaranteed by a denotational semantics.

The *Lambda Calculus of Objects* of [12] is an untyped λ-calculus enriched with object primitives. Objects are built up from an *empty object* by adding new methods or overriding existing ones. A primitive call to the methods of the objects is provided. The calculus supports a simple inheritance mechanism, a straightforward *mytype* method specialization, and dynamic lookup of methods. Its operational semantics deals with the special symbol *self* of object-oriented languages directly by lambda abstraction. This calculus, however: (i) lacks of a subtyping relation on objects; (ii) consider the objects as *ordered sequences* of methods instead of *sets* of methods (apart from making difficult to write mutually recursive methods, this constraint leads to a somewhat complicated formulation of the operational semantics which makes use of a *bookeeping* reduction to extract the appropriate method upon the evaluation of a message); (iii) does not have an equational theory on objects.

[14] extends [12] with a the new `pro`-type, denoted by $\mathtt{pro}\,t.\langle\!\langle \mathtt{m}_i : \sigma_i \rangle\!\rangle^{i \in I}$, in order to add subtyping. If we can assign a `pro`-type to an object, then we can add new methods or override existing ones. At this level, only trivial subtyping is

possible. Then we can "change" the object into a different kind of object where methods cannot be altered (i.e. the only operation on objects is message sending), by "sealing" a `pro`-type into a real object-type denoted by $\mathtt{obj}\, t.\langle\!\langle \mathtt{m}_i : \sigma_i \rangle\!\rangle^{i \in I}$. Even if from the outside of the object the only operation is message sending, the internal methods can override other methods of their host object. Preventing from the outside extension and override gives (*self-covariant*) "width-depth" subtyping.

The [14] calculus and the $\mathcal{O}bj1^{+}_{<:}$ calculus are closely related. In fact, both calculi have two kinds of object-types. The `pro`-types can be compared with our diamond-types: the former does not allows subtyping, whereas the latter it does. Objects assigned to both types can be extended and overridden. Moreover the `obj`-types of [14] can be like our saturated object-types; the former agrees with a "width-depth" subtyping relation, whereas the latter allows only "width" subtyping. Objects assigned to `obj`-types cannot be extended nor overridden, whereas objects assigned to saturated types can be overridden. The next table compares the two calculi.

	`pro`-types	`obj`-types	diamond-types	saturated-types
Self-types	$\surd$	$\surd$		
Method extension	$\surd$		$\surd$	
Method override	$\surd$		$\surd$	$\surd$
Width-<:		$\surd$	$\surd$	$\surd$
Depth-<:		$\surd$		

In [10], an orthogonal solution was taken in order to add subtyping to the Lambda Calculus of Objects: a subtyping relation "compatible" with method extension was introduced. Subtyping is subject to the restriction that a method can be forgotten only if the remaining methods in the object do not refer to it. This is obtained by *labeling* the type of a method `m` by the names of the methods of the object that `m` uses. For example, we can derive for the (diagonal) `point` of Section 3 the type $\vdash \mathtt{point} : [\mathtt{x} : nat, \mathtt{y} : int_x]$, but we cannot derive for `point` the type $\vdash \mathtt{point} : [\mathtt{y} : int_x]$, since the method `y` uses `x`. As pointed out in [8, 17], there are programs which can be typed in [10] and cannot be typed in [14] and vice-versa.

[18] presents an "explicitly typed" version of the Lambda Calculus of Objects, by making use of *dynamic typing*. This calculus has a sound and decidable type system, "width" subtyping on labeled object-types, and it allows for *first-class* method bodies that can be passed as function arguments. This increase the expressiveness of the language, since it allows to write "portable methods".

In [9], a more flexible typing system for the Lambda Calculus of Objects is given, by allowing objects to be typed independently from the order of their method additions. This extension also gives provision for method invocation when the receiver of the message is an *incomplete object*, i.e. an object whose implementation is only partially specified. A permutation rewriting rule between methods is sound but no subtyping is provided for this calculus.

[8] contains a very clear and simple encoding of object-types, by combining bounded quantification with labeled-types; in fact, labels record not only the useful methods which are sent to or overridden in *self*, but also the transitive closure of (the dependencies of) the method used by *self* in the method body. This calculus features "width" subtyping on labeled object-types.

[7] shows that the *matching* relation [11] can be fruitfully be employed in the Lambda Calculus of Objects, by making a substantially simplification of the typing rules of [12].

Another related paper is [21], which combines row-variables and refined subtyping in presence of extensible objects. There are similarities with our proposal, in particular diamond-types behave like `Pre`- and `Maybe`-types of [21]. But the subtyping of [21] is weaker than ours, since, for example, one cannot derive that the type of "colored point" is less than the type of "point", i.e. using our notation, that $[\mathtt{x}{:}int, \mathtt{col}{:}colors]{<:}[\mathtt{x}{:}int]$. The reason of this weackness is due to the fact that, in [21] the only subtyping rules are $(Shift_\diamond)$ (that convert a `Pre`-type into a `Maybe`-type), and $(Extend_\diamond)$ (that introduces a `Maybe`-type). Other differences are that we do not require object types to be total functions from names to types, and that we avoid row-variables by taking advantage of subtyping.

7.1 Conclusions

We presented an extension of the Calculus of Primitive Objects of [5], called $\mathcal{O}bj1^{+}_{\prec:}$, which allows one to dynamically add methods, and we introduced a static type system for this calculus that makes provision both for objects extension and for a "width" subtyping relation between object-types. The new features are obtained by extending the object-types of [5] with subsumption-parts, which convey information about methods that are subsumed. The $\mathcal{O}bj1^{+}_{\prec:}$ calculus allows a considerable number of programs to be typed, whereas they are not typable in $\mathcal{O}bj1_{\prec:}$, i.e. the original first-order system. The type systems allow for static detection of run-time errors such as *message-not-understood*.

A final remark concerns method encapsulation via variance annotations, a feature that is not accounted in our system and it is instead provided in [3]. However, the solution proposed in [3] could be accomplished as well as in the $\mathcal{O}bj1^{+}_{\prec:}$ calculus.

Moreover, the $\mathcal{O}bj1^{+}_{\prec:}$ type system can be easily extended with self-types by modeling the self-application semantics via *bounded universal polymorphism*. This conservative (w.r.t. $\mathcal{O}bj1^{+}_{\prec:}$) extension can be easily obtained with a very little cost with respect to the rules of $\mathcal{O}bj1^{+}_{\prec:}$, and it is presented in [16].

We conclude this paper with some open problems which will be subjects of future work:

1. Recently, M.Abadi has studied the possibility of extending the $\mathcal{O}bj1_{\prec:}$ calculus in an orthogonal way with respect to our $\mathcal{O}bj1^{+}_{\prec:}$ one. In this work-in-progress, a more flexible typing rule for method addition is given, by allowing incomplete objects to be typed independently from the order of their method additions. We believe this idea can be adopted and adapted with

labeled-types of [10]. This flexibility appears to be highly desirable for prototyping languages, such as delegation-based languages, where prototypes may reasonably be defined, and operated with as well, while part of their implementation (i.e. their methods) are yet to be defined. We also conjecture that our extension and Abadi's extension can be easily integrated, in order to build a calculus which allows for both features.
2. J. Palsberg [19] has described a (P-complete) type inference algorithm for an untyped version of the $\mathcal{O}bj1_{\prec:}$ calculus. Does this result holds also for the type inference version of $\mathcal{O}bj1^{+}_{\prec:}$?.
3. The type inference version of the Extended Primitive Calculus of Objects and the Lambda Calculus of Objects [12] share a lot of similarities. It seems reasonable to find a suitable encoding of one calculus into the other and a sound type system which fits both calculi.

Acknowledgments We wish to thank Martin Abadi and Luca Cardelli for their precious @-discussions, and the anonymous referees for their comments and suggestions.

References

1. M. Abadi. Baby Modula–3 and a Theory of Objects. *Journal of Functional Programming*, 4(2):249–283, 1994.
2. M. Abadi and L. Cardelli. A Theory of Primitive Objects: Second-Order Systems. *Science of Computer Programming*, 25(2-3):81–116, 1995.
3. M. Abadi and L. Cardelli. An Imperative Object Calculus. *Theory and Practice of Objects Systems*, 1(3):151–166, 1996.
4. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
5. M. Abadi and L. Cardelli. A Theory of Primitive Objects: Untyped and First Order System. *Information and Computation*, 125(2):78–102, 1996.
6. H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, Amsterdam, revised edition, 1984.
7. V. Bono and M. Bugliesi. Matching Constraint for the Lambda Calculus of Objects. In *Proc. of TLCA-97*, LNCS. Springer-Verlag, 1997. To appear.
8. V. Bono, M. Bugliesi, M. Dezani-Ciancaglini, and L. Liquori. Subtyping Constraint for Incomplete Objects. In *Proc. of CAAP-97*, LNCS. Springer-Verlag, 1997. To appear.
9. V. Bono, M. Bugliesi, and L. Liquori. A Lambda Calculus of Incomplete Objects. In *Proc. of MFCS-96*, volume 1113 of *LNCS*, pages 218–229. Springer-Verlag, 1996.
10. V. Bono and L. Liquori. A Subtyping for the Fisher-Honsell-Mitchell Lambda Calculus of Objects. In *Proc. of CSL-94*, volume 933 of *LNCS*, pages 16–30. Springer-Verlag, 1995.
11. K.B. Bruce, A. Shuett, and R. van Gent. Polytoil: a Type-safe Polymorphic Object-Oriented Language. In *Proc. of ECOOP-95*, volume 952 of *LNCS*, pages 16–30, 1995.
12. K. Fisher, F. Honsell, and J. C. Mitchell. A Lambda Calculus of Objects and Method Specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.
13. K. Fisher and J. C. Michell. The Development of Type Systems for Object Oriented Languages. *Theory and Practice of Objects Systems*, 1(3):189–220, 1995.

14. K. Fisher and J. C. Mitchell. A Delegation-based Object Calculus with Subtyping. In *Proc. of FCT-95*, volume 965 of *LNCS*, pages 42–61. Springer-Verlag, 1995.
15. A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.
16. L. Liquori. An Extended Theory of Primitive Objects. Technical Report CS-23-96, Computer Science Department, University of Turin, Italy, 1996.
17. L. Liquori. *Type Assigment Systems for Lambda Calculi and for the Lambda Calculus of Objects.* PhD thesis, University of Turin, February 1996.
18. L. Liquori and G. Castagna. A Typed Lambda Calculus of Objects. In *Proc. of Asian-96*, volume 1212 of *LNCS*, pages 129–141. Springer-Verlag, 1996.
19. J. Palsberg. Efficient Inference of Object Types. *Information and Computation*, 123:198–209, 1995.
20. G. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Aarhus University, Denmark, 1981.
21. D. Remy. Refined Subtyping and Row Variables for Record Types. Draft, 1995.

A The Typing Rules of $\mathcal{O}bj\mathbf{1}^{+}_{\prec:}$

Type Rules

$$\frac{}{\varepsilon \vdash ok}\ (Empty) \qquad \frac{\Gamma \vdash \sigma \qquad s \notin dom(\Gamma)}{\Gamma, s:\sigma \vdash ok}\ (Weak) \qquad \frac{\Gamma \vdash ok}{\Gamma \vdash \omega}\ (Type\!-\!\Omega)$$

$$\frac{\begin{array}{l}\Gamma \vdash \sigma_i \quad \forall i \in I \\ \Gamma \vdash \sigma_j \quad \forall j \in J \quad I \cap J = \emptyset\end{array}}{\Gamma \vdash [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]^{i \in I}_{j \in J}}\ (Diamond\!-\!Type) \qquad \frac{\Gamma \vdash \sigma_i \quad \forall i \in I}{\Gamma \vdash [\mathtt{m}_i : \sigma_i]^{i \in I}}\ (Sat\!-\!Type)$$

$$\frac{\Gamma, s:\sigma, \Gamma' \vdash ok}{\Gamma, s:\sigma, \Gamma' \vdash s:\sigma}\ (Proj) \qquad \frac{\Gamma \vdash \mathtt{o}:\sigma \qquad \Gamma \vdash \sigma <: \tau}{\Gamma \vdash \mathtt{o}:\tau}\ (<:)$$

(Let $\tau_i \triangleq [\mathtt{m}_h : \sigma_h]^{h \in H_i}$).

$$\frac{\Gamma, s_i:\tau_i \vdash \mathtt{o}_i : \sigma_i \qquad \forall i \in I \qquad H_i \subseteq I}{\Gamma \vdash [\mathtt{m}_i = \varsigma(s_i{:}\tau_i)\mathtt{o}_i]^{i \in I} : [\mathtt{m}_i : \sigma_i \diamond]^{i \in I}}\ (Object) \qquad \frac{\Gamma \vdash \mathtt{o} : [\mathtt{m}_k : \sigma_k]}{\Gamma \vdash \mathtt{o}.\mathtt{m}_k : \sigma_k}\ (Select)$$

$$\frac{\Gamma \vdash \mathtt{o}:\tau \qquad \Gamma \vdash \tau <: [\mathtt{m}_i : \sigma_i]^{i \in I} \qquad \Gamma, s_k : [\mathtt{m}_i : \sigma_i]^{i \in I} \vdash \mathtt{o}' : \sigma_k \qquad k \in I}{\Gamma \vdash \mathtt{o} \leftarrow\!\!\!\!\prec \mathtt{m}_k = \varsigma(s_k{:}[\mathtt{m}_i : \sigma_i]^{i \in I})\mathtt{o}' : \tau}\ (Over)$$

(Let $\tau_k \triangleq [\mathtt{m}_h : \sigma_h]^{h \in H}$).

$$\frac{\Gamma \vdash \mathtt{o} : [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]^{i \in I}_{j \in J} \qquad \Gamma, s_k : \tau_k \vdash \mathtt{o}' : \sigma_k \qquad H \subseteq I \qquad k \in J}{\Gamma \vdash \mathtt{o} \leftarrow\!\!\!\!\prec \mathtt{m}_k = \varsigma(s_k{:}\tau_k)\mathtt{o}' : [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]^{i \in I + \{k\}}_{j \in J - \{k\}}}\ (Ext)$$

Subtype Rules

$$\frac{\Gamma \vdash \sigma}{\Gamma \vdash \sigma <: \sigma}\ (Refl) \qquad \frac{\Gamma \vdash \sigma <: \tau \qquad \Gamma \vdash \tau <: \rho}{\Gamma \vdash \sigma <: \rho}\ (Trans) \qquad \frac{\Gamma \vdash \sigma}{\Gamma \vdash \sigma <: \omega}\ (\Omega)$$

$$\frac{\Gamma \vdash [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]_{j \in J}^{i \in I+K}}{\Gamma \vdash [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]_{j \in J}^{i \in I+K} <: [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]_{j \in J+K}^{i \in I}}\ (Shift_\diamond)$$

$$\frac{\Gamma \vdash [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]_{j \in J+K}^{i \in I}}{\Gamma \vdash [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]_{j \in J}^{i \in I} <: [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]_{j \in J+K}^{i \in I}}\ (Extend_\diamond)$$

$$\frac{\Gamma \vdash [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]_{j \in J}^{i \in I}}{\Gamma \vdash [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]_{j \in J}^{i \in I} <: [\mathtt{m}_i : \sigma_i]^{i \in I}}\ (Sat_\diamond)$$

$$\frac{\Gamma \vdash [\mathtt{m}_i : \sigma_i]^{i \in I+J}}{\Gamma \vdash [\mathtt{m}_i : \sigma_i]^{i \in I+J} <: [\mathtt{m}_i : \sigma_i]^{i \in I}}\ (Width)$$

Typed Equational Theory

$$\frac{\Gamma \vdash \mathsf{o}_2 \stackrel{ev}{=} \mathsf{o}_1 : \tau}{\Gamma \vdash \mathsf{o}_1 \stackrel{ev}{=} \mathsf{o}_2 : \tau}\ (Eq{-}Symm) \qquad \frac{\Gamma \vdash \mathsf{o}_1 \stackrel{ev}{=} \mathsf{o}_2 : \tau \qquad \Gamma \vdash \mathsf{o}_2 \stackrel{ev}{=} \mathsf{o}_3 : \tau}{\Gamma \vdash \mathsf{o}_1 \stackrel{ev}{=} \mathsf{o}_3 : \tau}\ (Eq{-}Trans)$$

$$\frac{\Gamma \vdash \mathsf{o}_1 \stackrel{ev}{=} \mathsf{o}_2 : \sigma \qquad \Gamma \vdash \sigma <: \tau}{\Gamma \vdash \mathsf{o}_1 \stackrel{ev}{=} \mathsf{o}_2 : \tau}\ (Eq{-}{<:}) \qquad \frac{\Gamma \vdash \mathsf{o}_1 : \sigma \qquad \Gamma \vdash \mathsf{o}_2 : \tau}{\Gamma \vdash \mathsf{o}_1 \stackrel{ev}{=} \mathsf{o}_2 : \omega}\ (Eq{-}\Omega)$$

$$\frac{\Gamma, s : \tau, \Gamma' \vdash ok}{\Gamma, s : \tau, \Gamma' \vdash s \stackrel{ev}{=} s : \tau}\ (Eq{-}Var) \qquad \frac{\Gamma \vdash \mathsf{o}_1 \stackrel{ev}{=} \mathsf{o}_2 : [\mathtt{m}_k : \sigma_k]}{\mathsf{o}_1.\mathtt{m}_k \stackrel{ev}{=} \mathsf{o}_2.\mathtt{m}_k : \sigma_k}\ (Eq{-}Select)$$

$$\frac{\begin{array}{ll}\Gamma \vdash \mathsf{o}_1 \stackrel{ev}{=} \mathsf{o}_2 : \tau & \Gamma \vdash \tau <: [\mathtt{m}_i : \sigma_i]^{i \in I} \\ \Gamma, s_k : [\mathtt{m}_i : \sigma_i]^{i \in I} \vdash \mathsf{o}' \stackrel{ev}{=} \mathsf{o}'' : \sigma_k & k \in I\end{array}}{\Gamma \vdash \mathsf{o}_1 \leftarrow\!\!- \mathtt{m}_k = \varsigma(s_k{:}[\mathtt{m}_i : \sigma_i]^{i \in I})\mathsf{o}' \stackrel{ev}{=} \mathsf{o}_2 \leftarrow\!\!- \mathtt{m}_k = \varsigma(s_k{:}[\mathtt{m}_i : \sigma_i]^{i \in I})\mathsf{o}'' : \tau}\ (Eq{-}Over)$$

(Let $\tau_k \triangleq [\mathtt{m}_h : \sigma_h]^{h \in H}$, and $\tau \triangleq [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]_{j \in J - \{k\}}^{i \in I + \{k\}}$).

$$\frac{\begin{array}{ll}\Gamma \vdash \mathsf{o}_1 \stackrel{ev}{=} \mathsf{o}_2 : [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]_{j \in J}^{i \in I} & H \subseteq I \\ \Gamma, s_k : \tau_k \vdash \mathsf{o}' \stackrel{ev}{=} \mathsf{o}'' : \sigma_k & k \in J\end{array}}{\Gamma \vdash \mathsf{o}_1 \leftarrow\!\!- \mathtt{m}_k = \varsigma(s_k{:}\tau_k)\mathsf{o}' \stackrel{ev}{=} \mathsf{o}_2 \leftarrow\!\!- \mathtt{m}_k = \varsigma(s_k{:}\tau_k)\mathsf{o}'' : \tau}\ (Eq{-}Ext)$$

Typed Equational Theory (continue)

(Let $\tau_i \triangleq [\mathtt{m}_h : \sigma_h]^{h \in H_i}$, and $\tau_j \triangleq [\mathtt{m}_k : \sigma_k]^{k \in K_j}$, and $\tau \triangleq [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]_{j \in J}^{i \in I}$).

$$\frac{\begin{array}{lll} \Gamma, s_i : \tau_i \vdash \mathsf{o}_i : \sigma_i & \forall\, i \in I & H_i \subseteq I \\ \Gamma, s_j : \tau_j \vdash \mathsf{o}_j : \sigma_j & \forall j \in J & K_j \subseteq I + J \end{array}}{\Gamma \vdash [\mathtt{m}_i = \varsigma(s_i{:}\tau_i)\mathsf{o}_i]^{i \in I} \stackrel{ev}{=} [\mathtt{m}_i = \varsigma(s_i{:}\tau_i)\mathsf{o}_i, \mathtt{m}_j = \varsigma(s_j{:}\tau_j)\mathsf{o}_j,]_{j \in J}^{i \in I} : \tau} \quad (Eq{-}Obj{-}{<:})$$

(In the next rules, let $\mathsf{o} \triangleq [\mathtt{m}_z = \varsigma(s_z{:}\tau_z)\mathsf{o}_z]^{z \in Z}$).

$$\frac{\Gamma \vdash \mathsf{o} : [\mathtt{m}_k : \sigma_k]}{\Gamma \vdash \mathsf{o}.\mathtt{m}_k \stackrel{ev}{=} \mathsf{o}_k\{s_k \leftarrow \mathsf{o}\} : \sigma_k} \quad (Eq{-}Select_{ev})$$

(Let $\rho \triangleq [\mathtt{m}_i : \sigma_i]^{i \in I}$).

$$\frac{\Gamma \vdash \mathsf{o} : \tau \qquad \Gamma \vdash \tau <: \rho \qquad \Gamma, s_k : \rho \vdash \mathsf{o}' : \sigma_k \qquad k \in I}{\Gamma \vdash \mathsf{o} \twoheadleftarrow \mathtt{m}_k = \varsigma(s_k{:}\rho)\mathsf{o}' \stackrel{ev}{=} [\mathtt{m}_z = \varsigma(s_z{:}\tau_z)\mathsf{o}_z, \mathtt{m}_k = \varsigma(s_k{:}\rho)\mathsf{o}']^{z \in Z - \{k\}} : \tau} \quad (Eq{-}Over_{ev})$$

(In the next rules, let $\tau_k \triangleq [\mathtt{m}_h : \sigma_h]^{h \in H}$, and $\tau \triangleq [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]_{j \in J - \{k\}}^{i \in I + \{k\}}$).

$$\frac{\Gamma \vdash \mathsf{o} : [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]_{j \in J}^{i \in I} \quad \Gamma, s_k : \tau_k \vdash \mathsf{o}' : \sigma_k \quad H \subseteq I \quad k \in Z - I}{\Gamma \vdash \mathsf{o} \twoheadleftarrow \mathtt{m}_k = \varsigma(s_k{:}\tau_k)\mathsf{o}' \stackrel{ev}{=} [\mathtt{m}_z = \varsigma(s_z{:}\tau_z)\mathsf{o}_z, \mathtt{m}_k = \varsigma(s_k{:}\tau_k)\mathsf{o}']^{z \in Z - \{k\}} : \tau} \quad (Eq{-}Ext^1_{ev})$$

$$\frac{\Gamma \vdash \mathsf{o} : [\mathtt{m}_i : \sigma_i \diamond \mathtt{m}_j : \sigma_j]_{j \in J}^{i \in I} \quad \Gamma, s_k : \tau_k \vdash \mathsf{o}' : \sigma_k \quad H \subseteq I \quad k \in J - Z}{\Gamma \vdash \mathsf{o} \twoheadleftarrow \mathtt{m}_k = \varsigma(s_k{:}\tau_k)\mathsf{o}' \stackrel{ev}{=} [\mathtt{m}_z = \varsigma(s_z{:}\tau_z)\mathsf{o}_z, \mathtt{m}_k = \varsigma(s_k{:}\tau_k)\mathsf{o}']^{z \in Z} : \tau} \quad (Eq{-}Ext^2_{ev})$$

This article was processed using the LaTeX macro package with LLNCS style