On Object Extension

Luigi Liquori

DIMI, Dip. Matematica ed Informatica, Università di Udine,
Via delle Scienze 206, I-33100 Udine, Italy
e-mail: liquori@dimi.uniud.it

Abstract. The last few years have seen the development of statically typed object based (also called prototype-based) programming languages. Two proposals, namely the *Lambda Calculus of Objects* of Fisher, Honsell, and Mitchell [15], and the *Object Calculus* of Abadi and Cardelli [2], have focused the attention of the scientific community on object calculi, as a foundation for the more traditional class-based calculi and as an original and safe style of programming. In this paper, we apply four type systems to the functional Lambda Calculus of Objects: (a) the Original type system [15]; (b) the Fisher's Ph.D type system [14]; (c) the Bruce's *Matching-based* type systems of Bono and Bugliesi [4], and (d) of Liquori [20]. We then compare these type systems with respect to the following points:

- small-step versus big-step semantics;
- implicit versus explicit polymorphism;
- Curry style versus Church style;
- static type checking versus run-time type checking;
- object extension and/or binary methods versus object subsumption (short account).

Categories. Type Systems of object-oriented languages (panorama).

1 Introduction

In this paper we present the functional Lambda Calculus of Objects of [15]. In its simplest version à la Curry[1], it is essentially an untyped lambda calculus enriched with three primitive operations on objects: *method addition* to define new methods, *method override* to redefine existing methods, and *method call* to send a message to (i.e. invoke a method on) an object. The calculus is simple and powerful enough to capture the class-based paradigm, since classes can be easily codified by appropriate objects, following the "classes-as-objects" analogy of Smalltalk-80 [18].

In the calculus of [15], objects can be seen as sequences (i.e. lists) of pairs *(method names, method bodies)* where the method body is (or reduces to) a lambda abstraction whose first formal parameter is always *self*. This calculus can be given an operational semantics which, in the case of a message send

[1] By *à la Curry*, we mean that the terms of the calculus are not annotated with types. This does not signify that a type system does not exist.

(written as $e \Leftarrow m$), produces the so-called *dynamic method lookup* in order to inspect the structure of objects and perform method extraction. This semantics can be given in terms of a transition (i.e. *small-step*) semantics, or in terms of an evaluation (i.e. *big-step*) semantics. To this calculus four static and sound type systems are applied, which enable us to prevent the unfortunate *message-not-found* run time error:

- the original type system of [15];
- the Fisher's Ph.D type system [14];
- the Bruce's *Matching-based* type system of Bono and Bugliesi [4];
- the Bruce's *Matching-based* type system of Liquori [20].

All of these solutions reinterpret the type of the *self* (or *this* in C^{++}), occurring inside a method body, in the type of the object which inherits that method; this capability is usually referred to as *mytype method specialization*.

For each of these solutions, we analyze how method specialization takes place and we show how the different object extension rules enable us to build polymorphic methods that work (hopefully) for all future extensions of the prototype (here the word *prototype* means the object we are extending or overriding).

The presentation is intentionally kept informal, with few definitions, no full type systems in appendix and no theorems. Quite simply, the main aim of this paper is to explain and compare existing systems concerning object extension.

We then take the step of enriching the above calculi with *explicit polymorphism*, in order to make our method bodies first-class values.

As a final step, we try to build a corresponding calculus à la Church[2], to which apply the four type systems described. As we will see, the progression from a totally untyped calculus to a fully decorated one is a not trivial task; in fact, one may need to resort to a *type-driven*[3] operational semantics, in the style of [13,11].

We finally build a calculus based on the Object Calculus of [2], where ordinary (i.e. fixed size) objects are extendible. This calculus considers the dynamic method lookup phase as an *implicit* phase that can be performed in just one step; in fact it eliminates all the rules concerning the method search from the operational semantics and the type systems (that are made explicit in [15]). Moreover, it includes as part of its syntax the lambda calculus (that was, instead, simulated by suitable objects in [2]).

We also compare this "hybrid" calculus with the Lambda Calculus of Objects: this comparison is interesting since objects in [2] are sets consisting of pairs *(method name, method body)* and, as it is customary in programming, sets can usually be implemented with lists.

The paper is organized as follows. In Section 2, we present the Lambda Calculus of Objects in Curry style together with a small-step and a big-step operational semantics. In Section 3 we present, quite informally, four type systems

[2] By *à la Church*, we mean that the terms of the calculus are annotated with types.

[3] By type-driven, we mean that some evaluation steps are constrained by suitable typing derivations.

with their *golden* rules of object extension. A comparison between these systems is given in Subsection 3.6. In Section 4, we try to build a corresponding calculus in Church style: we choose to study the type system of [14], since it is much more "plug-and-play" than other ones. Section 5 defines the calculus based on the Object Calculus of [2] and compares it with the Lambda Calculus of Object. Finally, Section 6 deals (concisely) with issues concerning the cohabitation of object extension and/or binary methods with object subsumption.

2 The Lambda Calculus of Objects à la Curry

In this section, we present the syntax and the dynamic semantics of the Lambda Calculus of Objects. The expressions are defined by the following grammar:

$$
\begin{array}{ll}
e ::= c \mid x \mid \lambda x.e \mid e_1\, e_2 \mid & \text{(untyped } \lambda\text{-calculus)} \\
\quad\;\; \langle\,\rangle \mid e \Leftarrow m \mid \langle e_1 \longleftarrow m = e_2 \rangle \mid \langle e_1 \leftarrow\!\!+\, m = e_2 \rangle \mid & \text{(object expressions)} \\
\quad\;\; e \hookleftarrow m, & \text{(auxiliary expression)}
\end{array}
$$

where c is a constant, x is a variable, and m is a method name. The object expressions have the following intuitive meaning:

- $\langle\,\rangle$ is the empty object;
- $e \Leftarrow m$ send message m to object e;
- $\langle e_1 \leftarrow\!\!+\, m = e_2 \rangle$ extend e_1 with a new method m;
- $\langle e_1 \longleftarrow m = e_2 \rangle$ replace the existing body of m in e_1 with body e_2.

Observe that the notation for methods and fields is unified. The auxiliary expression $e \hookleftarrow m$ searches the body of the m method within the object e; this form is mainly used to define the operational semantics and, in practice, is not available to the programmer[4]. The employment of the "search" expression is peculiar to the use of a different reduction semantics for the calculus; this semantics, inspired by [24,3,6], provides a more direct *dynamic method lookup* than the *bookkeeping* reductions originally introduced in [15].

The body of a method is (or reduces to) a lambda abstraction whose first parameter is always *self* (i.e. $\lambda \mathit{self}.\ldots$); in fact, the operational semantics will reduce a message send to the application of the body of the method to the recipient of the message. Note that argument passing can be modeled via lambda application (i.e. $(e \Leftarrow m)\ arg_1 \ldots arg_n$, with $n \geq 0$).

2.1 Operational Semantics

In this subsection we present a small-step reduction semantics and a big-step operational semantics.

[4] If a program is allowed to use such operator, then it breaks object encapsulation since the state of the object and the methods implementation are usually hidden from the outside.

Small-step Reduction Semantics. The core of the small-step reduction semantics is given by the following reduction rules. Let $\longleftarrow\!\!\!\!\ast$ denote either $+\!\!\!\!\!\longleftarrow$ or $\longleftarrow$.

$$
\begin{array}{llll}
(\mathit{Beta}) & (\lambda x.e_1)\, e_2 & \stackrel{ev}{\to} [e_2/x]e_1 & \\
(\mathit{Select}) & e \Leftarrow m & \stackrel{ev}{\to} (e \hookleftarrow m)\, e & \\
(\mathit{Succ}) & \langle e_1 \longleftarrow\!\!\!\!\ast\; m = e_2 \rangle \hookleftarrow m & \stackrel{ev}{\to} e_2 & \\
(\mathit{Next}) & \langle e_1 \longleftarrow\!\!\!\!\ast\; n = e_2 \rangle \hookleftarrow m & \stackrel{ev}{\to} e_1 \hookleftarrow m & m \neq n.
\end{array}
$$

In addition to the standard (Beta) rule for lambda calculus expressions, the main operation on objects is method invocation, whose reduction is defined by the (Select) rule: the result of sending a message m to an object e containing an m method is the result of self-applying the body of m to the object e itself. To account for this behavior, the (Succ) and (Next) reduction rules recursively inspect the structure of the object (implemented as a list) and perform method extraction; by looking at these last two rules, it follows that the $\hookleftarrow$ operator is *destructive*, i.e. it goes "through" the object until it finds the searched method. This will also be semantically enforced in the (search) type rules (see Section 3). We could then define the relation $\stackrel{ev}{\to}^{*}$ as the symmetric, reflexive, transitive and contextual closure of $\stackrel{ev}{\to}$. As it is customary, the reduction semantics does not specify an evaluation strategy which is forced, instead, in a big-step operational semantics.

Big-step Operational Semantics. We define an operational semantics via a natural proof deduction system à la Plotkin [27]. The purpose of the reduction is to map every closed expression into a normal form, i.e. an irreducible term. The strategy is "lazy" since it does not work under lambda binders and inside object expressions. We define the set of results (i.e. values) as follows:

$$
\begin{aligned}
\mathit{obj} &::= \langle e_1 \longleftarrow\!\!\!\!\ast\; m = e_2 \rangle \mid \langle\,\rangle \\
v &::= c \mid \mathit{obj} \mid \lambda x.e.
\end{aligned}
$$

The deduction rules are presented as follows:

$$
\frac{}{v \Downarrow v}\ (\mathit{Red}{-}\mathit{Val}) \qquad \frac{e_1 \Downarrow \lambda x.e_1' \qquad [e_2/x]e_1' \Downarrow v}{e_1\, e_2 \Downarrow v}\ (\mathit{Red}{-}\mathit{Beta})
$$

$$
\frac{e \Downarrow \mathit{obj} \qquad \mathit{obj} \hookleftarrow m \Downarrow \lambda x.e' \qquad [\mathit{obj}/x]e' \Downarrow v}{e \Leftarrow m \Downarrow v}\ (\mathit{Red}{-}\mathit{Select})
$$

$$
\frac{e_2 \Downarrow v}{\langle e_1 \longleftarrow\!\!\!\!\ast\; m = e_2 \rangle \hookleftarrow m \Downarrow v}\ (\mathit{Red}{-}\mathit{Succ})
$$

$$
\frac{e_1 \Downarrow \mathit{obj} \qquad \mathit{obj} \hookleftarrow m \Downarrow v \qquad m \neq n}{\langle e_1 \longleftarrow\!\!\!\!\ast\; n = e_2 \rangle \hookleftarrow m \Downarrow v}\ (\mathit{Red}{-}\mathit{Next})
$$

Big-step operational semantics is deterministic, and immediately suggests how to build an interpreter for the calculus. Moreover, big-step semantics is sound with respect to the reduction $\overset{ev}{\to}^{*}$, since it holds:

Proposition 1 (Soundness of $\Downarrow$). *If $e \Downarrow v$, then $e \overset{ev}{\to}^{*} v$.*

Definition 2. Given a closed expression e, we say that e *converges*, and we write it as $e \Downarrow$, if there exist a v such that $e \Downarrow v$.

Given the above definition, we also conjecture the completeness, which shows that every terminating program also terminates in our interpreter.

Conjecture 3 (Completeness of $\Downarrow$). *If $e \overset{ev}{\to}^{*} v$, then e converges, i.e. $e \Downarrow$.*

2.2 Some Intuitive Examples

Let $\langle m_1{=}e_1 \ldots m_k{=}e_k \rangle$ be as shorthand for $\langle \ldots \langle \langle \rangle \leftarrow\!\!+ m_1 = e_1 \rangle \ldots \leftarrow\!\!+ m_k = e_k \rangle$ for $k \geq 1$.

Example 4 (Method Dependencies). This very simple example will follow us through the presentation of the type systems: it will help us to highlight how method dependencies are carried out in the systems. The object

$$e \triangleq \langle m = \lambda self.1, n = \lambda self.self \Leftarrow m \rangle,$$

represents a point with two methods m and n, where n gives the same result as m. Moreover, we find it useful to consider the following objects:

$$\begin{aligned} e' &\triangleq \langle e \leftarrow\!\!+ p = \lambda self.self \Leftarrow n \rangle \\ e'' &\triangleq \langle e \leftarrow\!\!+ q = \lambda self.self \Leftarrow n \rangle \\ e''' &\triangleq \langle l = \lambda self.1, n = \lambda self.self \Leftarrow l, q = \lambda self.self \Leftarrow n \rangle. \end{aligned}$$

Example 5 (A "funny" object). Let the object *funny* be defined as follows:

$$funny \triangleq \langle m = \lambda self.\langle self \longleftarrow m = \lambda self'.self' \Leftarrow m \rangle \rangle.$$

This is an object whose evaluation may be infinite. If we send the message m to *funny*, then we have the following computation:

$$\begin{aligned} funny \Leftarrow m &\overset{ev}{\to} (funny \hookleftarrow m)\, funny \\ &\overset{ev}{\to} (\lambda self.\langle self \longleftarrow m = \lambda self'.self' \Leftarrow m \rangle) funny \\ &\overset{ev}{\to} \langle funny \longleftarrow m = \lambda self'.self' \Leftarrow m \rangle. \end{aligned}$$

Conversely, if we send the message m to *funny* twice, then the computation becomes infinite:

$$
\begin{aligned}
(\mathit{funny} \Leftarrow m) \Leftarrow m &\stackrel{ev}{\to}{}^{*} \langle \mathit{funny} \longleftarrow m = \lambda \mathit{self}'.\mathit{self}' \Leftarrow m\rangle \Leftarrow m \\
&\stackrel{ev}{\to} (\lambda \mathit{self}'.\mathit{self}' \Leftarrow m)\langle \mathit{funny} \longleftarrow m = \lambda \mathit{self}'.\mathit{self}' \Leftarrow m\rangle \\
&\stackrel{ev}{\to} \langle \mathit{funny} \longleftarrow m = \lambda \mathit{self}'.\mathit{self}' \Leftarrow m\rangle \Leftarrow m \\
&\stackrel{ev}{\to}{}^{*} \langle \mathit{funny} \longleftarrow m = \lambda \mathit{self}'.\mathit{self}' \Leftarrow m\rangle \Leftarrow m \\
&\stackrel{ev}{\to}{}^{*} \ldots
\end{aligned}
$$

Since *funny* is typable in all the type systems to be presented, it illustrates the failure of strong normalization for typable object expressions.

3 The Four Type Systems

In this section we present the four type systems applied to the Lambda Calculus of Objects. In particular we show the syntax of types and contexts, the various judgments, and the main typing rules: for the purpose of the comparison between systems, we only are interested in discussing the rules of method extension, method override, message send and method search.

Typing the Examples The objects presented in Examples 4, and 5 can be typed *in all the four type systems* as follows:

$$
\begin{aligned}
&\varepsilon \vdash e : \mathtt{obj}\, t.\langle\!\langle m : int, n : int\rangle\!\rangle \\
&\varepsilon \vdash e' : \mathtt{obj}\, t.\langle\!\langle m : int, n : int, p : int\rangle\!\rangle \\
&\varepsilon \vdash e'' : \mathtt{obj}\, t.\langle\!\langle m : int, n : int, q : int\rangle\!\rangle \\
&\varepsilon \vdash e''' : \mathtt{obj}\, t.\langle\!\langle l : int, n : int, q : int\rangle\!\rangle \\
&\varepsilon \vdash \mathit{funny} : \mathtt{obj}\, t.\langle\!\langle m : t\rangle\!\rangle.
\end{aligned}
$$

Note that, in the judgment for *funny*, the method m has a type t which refers to the type of the object *funny* itself.

Remark 6. All the considered type systems do not have a *subsumption* rule of the shape:

$$\frac{\Gamma \vdash e : \sigma \qquad \Gamma \vdash \sigma \ \mathit{ord}\ \tau}{\Gamma \vdash e : \tau} \quad (\mathit{subsume})$$

where *ord* is any partial order on types. The issue of object subsumption in presence of object extension has been widely studied [6,17,2,14,5,21,20,28]. A detailed comparison of calculi with object extension in presence of object subsumption is under development (see also Section 6).

3.1 The Original Type System of [15]

Definition 7 (Type Syntax). The set of types, rows and kinds are mutually defined by the following grammar:

$$\begin{array}{ll} \text{Types } \tau & ::= t \mid \tau \to \tau \mid \texttt{obj}\, t.R \\ \text{Rows } R & ::= r \mid \langle\!\langle\rangle\!\rangle \mid \langle\!\langle R \mid m : \tau \rangle\!\rangle \mid \lambda t.R \mid R\,\tau \\ \text{Kinds } \kappa & ::= T \mid T^p \to [m_1, \ldots, m_k] \ (p \geq 0, k \geq 1). \end{array}$$

The binder `obj` is a sort of fixed-point operator that scopes over the row-part: the bound type-variable t may occur freely within the scope of the binder, with every free occurrence referring to the object itself, i.e. *self*. Thus, object-types are a form of recursively-defined types. A *row* R is an unordered collection of pairs *(method label, method type)*; we consider α-conversion of type-variables bound by `obj`. Additional equations between types and rows arise as a result of β-reduction. Intuitively, if an object-type $\texttt{obj}\, t.\langle\!\langle m_1 : \tau_1 \ldots m_k : \tau_k \rangle\!\rangle$ with $k \geq 0$ is assigned to an object e, then e can receive $m_1 \ldots m_k$ messages, and the final result types are $\tau_1 \ldots \tau_k$. We write $\overline{m} : \overline{\tau}$ to abbreviate $m_1 : \tau_1, \ldots, m_k : \tau_k$, for some unimportant k.

Contexts and Type Judgments. Contexts are ordered lists (not sets) of the following shape:

$$\Gamma ::= \varepsilon \mid \Gamma, x : \tau \mid \Gamma, t : T \mid \Gamma, r : \kappa,$$

and judgments have the following form:

$$\Gamma \vdash *, \text{ or } \ \Gamma \vdash R : \kappa, \text{ or } \ \Gamma \vdash \tau : T, \text{ or } \ \Gamma \vdash e \ : \ \tau.$$

The judgment $\Gamma \vdash *$ can be read as "Γ is a well-formed context". Intuitively, the meaning of the judgment $\Gamma \vdash R : [m_1, \ldots, m_k]$ assures that the row R does *not* include method names $m_1, \ldots, m_k$. For example:

$$\varepsilon \vdash \langle\!\langle m : int, n : int \rangle\!\rangle : [p],$$

being that $m \neq n \neq p$. We need this *negative* information to guarantee statically that methods are not multiply defined. When $\Gamma \vdash R : T \to [\overline{m}]$, then it follows that R must be a row-abstraction, e.g.:

$$\varepsilon \vdash \lambda t.\langle\!\langle \overline{n} : \overline{\tau} \rangle\!\rangle : T \to [\overline{m}],$$

being that $\overline{n} \neq \overline{m}$. The meaning of the other judgments is the standard one (i.e. τ is a "well-formed context" in Γ, and τ is assigned to e in the context Γ).

Main Typing Rules. The empty object $\langle\,\rangle$ has the object-type $\texttt{obj}\, t.\langle\!\langle\rangle\!\rangle$: this object cannot respond to any message, but can be extended with other methods. The typing rule is:

$$\frac{\Gamma \vdash *}{\Gamma \vdash \langle\,\rangle : \texttt{obj}\, t.\langle\!\langle\rangle\!\rangle} \quad (empty{-}obj)$$

The rule to give a type to a message send is simple:

$$\frac{\Gamma \vdash e : \texttt{obj}\, t.\langle\!\langle R \mid n : \tau \rangle\!\rangle}{\Gamma \vdash e \Leftarrow n : [\texttt{obj}\, t.\langle\!\langle R \mid n : \tau \rangle\!\rangle / t]\tau} \quad (send)$$

This rule says that we can give a type to a message send provided that the receiver has the method we require in its object-type. Since the order of methods in rows is irrelevant, we can write n as the last method listed in the object-type. The result of a message send will have a type in which every occurrence of the type-variable t has to be substituted by the full type of the object itself, thus reflecting the recursive nature of the object-type.

The most subtle and intriguing rule is the one that enable us to build another object by extending an existing prototype.

$$\frac{\begin{array}{ll}\Gamma \vdash e_1 \ : \ \texttt{obj}\, t.\langle\!\langle R \mid \overline{m} : \overline{\sigma} \rangle\!\rangle & \Gamma, t : T \vdash R \ : \ [\overline{m}, n] \\ \Gamma, r : T \to [\overline{m}, n] \vdash e_2 \ : \ [\texttt{obj}\, t.\langle\!\langle r\, t \mid \overline{m} : \overline{\sigma}, n : \tau \rangle\!\rangle / t](t \to \tau) & r \text{ not in } \tau\end{array}}{\Gamma \vdash \langle e_1 \leftarrow\!\!+\, n = e_2 \rangle \ : \ \texttt{obj}\, t.\langle\!\langle R \mid \overline{m} : \overline{\sigma}, n : \tau \rangle\!\rangle} \quad (obj{-}ext)$$

In this rule, we assume the type of the object e_1 does not contain the method n we want to add. This condition is guaranteed by the first two premises. The meaning of the explicitly listed methods $\overline{m}$ in the types of e_1 and e_2 is crucial:

- in the typing of e_2 they represent the methods which are *useful* to type n's body: i.e. (at least) the messages that are sent to *self* or the methods overridden to *self* inside the body of n[5].
- in the typing of e_1 they guarantee that the methods $\overline{m}$, which are useful to type the body of n, are *already* present in the prototype to be extended.

The side condition "r not in τ" is an "hygiene condition" that avoids to introduce unsound free occurrences of r. As an example, if the body e_2 of method n is:

$$body_n = \lambda self.\langle self \longleftarrow m_1 = \lambda self'.self' \Leftarrow m_2 \rangle,$$

then the addition of n to an object e_1 (not containing n) requires the following judgment to be derivable:

$$r : T \to [m_1, m_2, n] \vdash body_n : [\texttt{obj}\, t.\langle\!\langle r\, t \mid m_1 : \_1, m_2 : \_2, n : t \rangle\!\rangle / t](t \to t),$$

for some unknown types $\_1$, and $\_2$, such that $\_1 = \_2$[6].

Self-Application. Note that the typing of e_2 is an arrow-type of the shape $(t \to \tau)$ with t substituted by an object-type. Since t is hidden in the final typing of $\langle e_1 \leftarrow\!\!+\, n = e_2 \rangle$, it is necessary in the typing of e_2, because the semantics of sending messages would result in the application of the body of the method to the host object itself.

[5] Cardelli [10] defines the capability of a method of operating directly on its own self as a *self-inflicted* operation.

[6] In UNIX jargon, in order to type an object extension, we need to be able to `grep` all the $\overline{m}$ methods that are essential to the typing of the body of n (plus n itself, to guarantee recursive method definition).

Higher-Order. Note also that the typing for e_2 contains occurrences of the "open" object-type $\texttt{obj}\, t.\langle\!\langle r\, t \mid \overline{m} : \overline{\sigma}, n : \tau\rangle\!\rangle$. Inside that object-type there occurs an application of the row-variable r (which is implicitly quantified in the context) to the type t (representing the type of *self*). Because of this implicit quantification, for every substitution of r with a row R' of the same kind of r, e_2 will have the indicated type in which r will be substituted by R'. This guarantees that for all future extensions of the object $\langle e_1 \leftarrow\!\!+\, n = e_2\rangle$[7], the body e_2 of n will specialize its functionality. The "very high mutability" of the type of a method will be very useful when we introduce *explicit quantification* for method bodies (see Subsection 3.7).

The rule that allows to build another object by overriding a method which already belongs to the prototype is as follows:

$$\frac{\begin{array}{l}\Gamma \vdash e_1 \;:\; \texttt{obj}\, t.\langle\!\langle R \mid \overline{m} : \overline{\sigma}, n : \tau\rangle\!\rangle \\ \Gamma, r : T \to [\overline{m}, n] \vdash e_2 \;:\; [\texttt{obj}\, t.\langle\!\langle r\, t \mid \overline{m} : \overline{\sigma}, n : \tau\rangle\!\rangle / t](t \to \tau)\end{array}}{\Gamma \vdash \langle e_1 \longleftarrow n = e_2\rangle \;:\; \texttt{obj}\, t.\langle\!\langle R \mid \overline{m} : \overline{\sigma}, n : \tau\rangle\!\rangle} \quad (obj\!-\!over)$$

The first premise says that the method $n : \tau$ we are overriding and the methods $\overline{m} : \overline{\sigma}$ which are useful to type the e_2 body are present in the type of e_1. The second premise is as in the $(obj\!-\!ext)$ rule. Note that the type of the overridden method is left unchanged, and that the type of the new object is the same as the one of the prototype.

Finally the rule of method search is as follows (this rule does not pertain to the type system of [15], since a different operational semantics is adopted that uses the "bookkeeping" reduction rules to extract the appropriate method out of an object):

$$\frac{\Gamma \vdash e \;:\; \texttt{obj}\, t.\langle\!\langle R \mid n : \tau\rangle\!\rangle \qquad \Gamma, t : T \vdash \langle\!\langle R \mid \overline{m} : \overline{\sigma}\rangle\!\rangle : [n]}{\Gamma \vdash e \hookleftarrow n \;:\; [\texttt{obj}\, t.\langle\!\langle R \mid \overline{m} : \overline{\sigma}, n : \tau\rangle\!\rangle / t](t \to \tau)} \quad (search)$$

The first premise of this rule says that that the object e contains n in its interface, while the second premise "attaches" to object e all the methods which where skipped during the right-to-left traversing of object e' (built using e as a prototype), which received the message n in question. Hence, the final type for $e \hookleftarrow n$ will have the same functionality of the body of n, i.e. an arrow-type whose first parameter has the type of e' i.e. $\texttt{obj}\, t.\langle\!\langle R \mid \overline{m} : \overline{\sigma}, n : \tau\rangle\!\rangle$.

As an important remark, we note that there is no way of finding the $\overline{m} : \overline{\sigma}$ methods; in fact, the search operator is "destructive" and does not keep track of the already skipped method. This rule only guarantees that the body of n will specialize its functionality for all future extensions of e.

Example 8 (Typing Example 4 in [15]). In order to build the object e', using e as a prototype, we need (i) to know that p does not belongs to e, (ii) to assure

[7] Much more precisely: all prototypes containing the $\overline{m} : \overline{\sigma}$ methods, and not containing the n method, can be extended with $n = e_2$, since the type of n specializes its functionality.

that $n : int$ is present in the the type of e, and (iii) to derive for the body of p the judgment

$$r : T{\to}[n,p] \vdash \lambda self.self \Leftarrow n : [\mathtt{obj}\, t.\langle\!\langle r\, t \mid n : int, p : int\rangle\!\rangle/t](t{\to}int). \qquad (*)$$

It follows that, to type the body of p, we need to know only the types of methods n (the method directly used inside p), and p (i.e. the method to be added); there is no need to extract the indirect dependencies of n (i.e. m). Of course one can also add indirect dependencies, but these are not essential to the method specialization of p. This form of polymorphism is quite powerful for two reasons:

- the body of the added method has a very high "degree of polymorphism", since a very small amount of information is needed in order to give a correct type for the body of p. Method specialization of p will follow for all future extensions of e'.
- the body of p should also be used to extend other objects such as the e'' or e''', using the same judgment $(*)$.

For the sake of curiosity, the object *funny* can be typed using the following judgment:

$$r : T{\to}[m] \vdash \lambda self.\langle self \longleftarrow m = \lambda self'.self' \Leftarrow m\rangle : [\mathtt{obj}\, t.\langle\!\langle r\, t \mid m : t\rangle\!\rangle/t](t{\to}t).$$

3.2 Fisher's Type System [14]

The set of types, rows and kinds are defined exactly as in Definition 7.

Contexts and Type Judgments. Contexts have the following shape:

$$\Gamma ::= \varepsilon \mid \Gamma, x : \tau \mid \Gamma, t : T \mid \Gamma, r <: R : \kappa,$$

and judgments have the following form:

$$\Gamma \vdash *, \text{ or } \Gamma \vdash R : \kappa, \text{ or } \Gamma \vdash \tau : T, \text{ or } \Gamma \vdash e : \tau, \text{ or } \Gamma \vdash R <: R'.$$

With respect to contexts and judgments of [15], there are some differences:

- the judgment $\Gamma \vdash R <: R'$ formalizes *with subrowing*, i.e. row R has *more* methods than row R', and common methods have the same type.
- the declaration $r <: R : \kappa$ occurring in a context gives us some kind of *positive* information about the possible shape of the rows to be substituted with r inside an object-type. More precisely, the row-variable r must be a *subrow* of the explicitly listed row R. The kind κ still continues to have the same meaning as in [15].

The meaning of the other judgments is the same as in [15].

Main Typing Rules. The type rule to build an empty object is the same as in [15].

The rule to give a type to a message send is as follows:

$$\frac{\Gamma \vdash e : \mathtt{obj}\, t.R \qquad \Gamma, t : T \vdash R <: \langle\!\langle n : \tau \rangle\!\rangle}{\Gamma \vdash e \Leftarrow n : [\mathtt{obj}\, t.R/t]\tau} \quad (send)$$

This rule is rather different from the one of [15]; in fact, it requires that the row R occurring in the type of e contains (this is the "positive" information given by the subrowing judgment) the message n to be received. The object-type $\mathtt{obj}\, t.R$ could be also "open" i.e. of the shape $\mathtt{obj}\, t.r\, t$, or $\mathtt{obj}\, t.\langle\!\langle r\, t \mid \ldots \rangle\!\rangle$ (and so on), in order to type a message send to objects whose types may be partially abstract.

The rule to extend an object is as follows:

$$\frac{\begin{array}{ll} \Gamma \vdash e_1 \;:\; \mathtt{obj}\, t.R & \Gamma, t : T \vdash R \;:\; [n] \\ \Gamma, r <: \lambda t.\langle\!\langle R \mid n : \tau \rangle\!\rangle : T \to [\;] \vdash e_2 \;:\; [\mathtt{obj}\, t.r\, t/t](t \to \tau) & r \text{ not in } \tau \end{array}}{\Gamma \vdash \langle e_1 \leftarrow\!\!+\, n = e_2 \rangle \;:\; \mathtt{obj}\, t.\langle\!\langle R \mid n : \tau \rangle\!\rangle} \quad (obj{-}ext)$$

In this rule we require that the prototype e_1 does not contain the method n to be added (this is the "negative" information given by the "well kindness" judgment), and we require that the type of e_2 is an arrow-type of the shape $(t \to \tau)$ with t substituted by an open (i.e. still unknown) object-type of the shape $\mathtt{obj}\, t.r\, t$, being that $r <: \lambda t.\langle\!\langle R \mid n : \tau \rangle\!\rangle$. The constraint on the shape of r together with its kinding (i.e. $T \to [\;]$) are crucial in order to understand this rule:

- the positive information of the subrowing judgment gives us the information that for all future extensions of the object $\langle e_1 \leftarrow\!\!+\, n = e_2 \rangle$, the body e_2 of n will specialize its functionality.
- the negative information $T \to [\;]$ force that the body of n cannot self-inflict an object extension to the host object.

The rule for object override is simpler:

$$\frac{\begin{array}{l} \Gamma \vdash e_1 \;:\; \mathtt{obj}\, t.R \qquad \Gamma, t : T \vdash R <: \langle\!\langle n : \tau \rangle\!\rangle \\ \Gamma, r <: \lambda t.R : T \to [\;] \vdash e_2 \;:\; [\mathtt{obj}\, t.r\, t/t](t \to \tau) \end{array}}{\Gamma \vdash \langle e_1 \longleftarrow n = e_2 \rangle \;:\; \mathtt{obj}\, t.R} \quad (obj{-}over)$$

The only remark about this rule is that the subrowing judgment forces the method n in the row R.

Finally, the rule of method search is as follows (also in [14] the "bookkeeping" reduction rules are adopted):

$$\frac{\Gamma \vdash e \;:\; \mathtt{obj}\, t.R \qquad \Gamma, t : T \vdash R <: \langle\!\langle n : \tau \rangle\!\rangle \qquad \Gamma, t : T \vdash R' <: R}{\Gamma \vdash e \hookleftarrow n \;:\; [\mathtt{obj}\, t.R'/t](t \to \tau)} \quad (search)$$

The first two premises assure that n is a method of e, while the last premise finds a row R' that extends R. The final type for the host object will be $\texttt{obj}\, t.R'$. This type represents the receiver of message n in question.

As in the $(search)$ rule for [15], and due to the nature of the "search" operator, there is no way of finding a precise R' which extends R.

Example 9 (Typing Example 4 in [14]). In order to build the object e', using e as a prototype, we need (i) to know that p does not belongs to e, and (ii) to derive for the body of p the judgment:

$$r <: \lambda t.\langle\langle m : int, n : int, p : int\rangle\rangle : T \to [\,] \vdash \lambda self.self \Leftarrow n : [\texttt{obj}\, t.r\, t/t](t \to int). \ (*)$$

It follows that, in order to type the body of p, we need to know the types of methods m, n (i.e. *all methods of the prototype* e) and p itself.

This "degree of polymorphism" (lower with respect to [15]) is exactly what we need in order to capture method specialization of p for all future extensions of e'. Note that the judgment $(*)$ can be used to extend e'' but not e''', since, of course, e''' is not an extension of e.

3.3 Bruce's Matching-based Type System of Bono and Bugliesi [4]

This solution is well known to perform a substantial simplification of the original type system of [15].

Type Syntax. Types and rows have the following shape:

$$\begin{array}{ll} \text{Types } \tau & ::= t \mid \tau \to \tau \mid \texttt{obj}\, t.R \\ \text{Rows } R & ::= \langle\langle\rangle\rangle \mid \langle\langle R \mid m : \tau \rangle\rangle. \end{array}$$

Contexts and Type Judgments. Contexts have the following shape:

$$\Gamma ::= \varepsilon \mid \Gamma, x : \tau \mid \Gamma, u \mathrel{<\!\#} \sigma,$$

and the judgments have the following form:

$$\Gamma \vdash *, \text{ or } \ \Gamma \vdash \tau : T, \text{ or } \ \Gamma \vdash e \ : \ \tau, \text{ or } \ \Gamma \vdash \sigma \mathrel{<\!\#} \tau.$$

The relation of *Matching* [8], formalized by the judgment $\Gamma \vdash \sigma \mathrel{<\!\#} \tau$, captures the capability of an object to inherit the behavior and to specialize the types of the methods of the prototype.

The main rule underlining this relation is the following one:

$$\frac{\Gamma \vdash \texttt{obj}\, t.\langle\langle \overline{m} : \overline{\sigma}, \overline{n} : \overline{\tau} \rangle\rangle}{\Gamma \vdash \texttt{obj}\, t.\langle\langle \overline{m} : \overline{\sigma}, \overline{n} : \overline{\tau} \rangle\rangle \mathrel{<\!\#} \texttt{obj}\, t.\langle\langle \overline{m} : \overline{\sigma} \rangle\rangle} \ (\mathrel{<\!\#})$$

Hence, an object-type matches an other if and only if the former has more methods than the latter. Note that the type-variable u "match-bounded" in a context Γ always refers to the type of *self*. This condition is enforced in the typing rules.

Main Typing Rules. Again, the type rule for building an empty object is the same as in [15] and [14].

The rule to give a type to a message send behaves as follows:

$$\frac{\Gamma \vdash e : \sigma \qquad \Gamma \vdash \sigma <\!\# \, \texttt{obj}\, t.\langle\!\langle n : \tau \rangle\!\rangle}{\Gamma \vdash e \Leftarrow n : [\sigma/t]\tau} \quad (send)$$

In this rule we require that the type σ has the method n in its protocol; this can be achieved when σ is an object-type containing n, but also when the type σ of e is partially abstract (e.g. when σ is a type-variable match-bounded in the context Γ).

The type rule to extend an object is as follows:

$$\frac{\begin{array}{l}\Gamma \vdash e_1 \ : \ \texttt{obj}\, t.R \\ \Gamma, u <\!\# \, \texttt{obj}\, t.\langle\!\langle R \mid n : \tau \rangle\!\rangle \vdash e_2 \ : \ [u/t](t \rightarrow \tau) \quad n \text{ not in } R\end{array}}{\Gamma \vdash \langle e_1 \leftarrow\!\!+\, n = e_2 \rangle \ : \ \texttt{obj}\, t.\langle\!\langle R \mid n : \tau \rangle\!\rangle} \quad (obj{-}ext)$$

In this rule, as in all the rules for object extension we have previously seen, we require that e_1 does not contain the method n to be added (enforced by the side condition n not in R), and that the type of e_2 is an arrow type $(t \rightarrow \tau)$ with t substituted for a unknown type u, match-bounded by the object-type $\texttt{obj}\, t.\langle\!\langle R \mid n : \tau \rangle\!\rangle$. The implicit match-bound of the type-variable u (referring to the type of *self*) assures that e_2 specializes its functionality for all future extensions of $\langle e_1 \leftarrow\!\!+\, n = e_2 \rangle$.

The rule for object override is simpler:

$$\frac{\begin{array}{l}\Gamma \vdash e_1 \ : \ \sigma \qquad \Gamma \vdash \sigma <\!\# \, \texttt{obj}\, t.\langle\!\langle \overline{m} : \overline{\sigma}, n : \tau \rangle\!\rangle \\ \Gamma, u <\!\# \, \texttt{obj}\, t.\langle\!\langle \overline{m} : \overline{\sigma}, n : \tau \rangle\!\rangle \vdash e_2 \ : \ [u/t](t \rightarrow \tau)\end{array}}{\Gamma \vdash \langle e_1 \longleftarrow n = e_2 \rangle \ : \ \sigma} \quad (obj{-}over)$$

The only interesting remark is that σ could be a type-variable match-bounded in the context Γ: it follows that a self-inflicted object override is allowed (as well as in all other type systems) inside method bodies.

As a side remark, we conjecture the soundness of the type system by substituting the $(obj{-}over)$ with the following rule which better captures the "philosophy" of the matching as a protocol extension.

$$\frac{\begin{array}{l}\Gamma \vdash e_1 \ : \ \sigma \quad \Gamma \vdash \sigma <\!\# \, \texttt{obj}\, t.\langle\!\langle n : \tau \rangle\!\rangle \\ \Gamma, u <\!\# \, \sigma \vdash e_2 \ : \ [u/t](t \rightarrow \tau)\end{array}}{\Gamma \vdash \langle e_1 \longleftarrow n = e_2 \rangle \ : \ \sigma} \quad (obj{-}over')$$

Finally the rule of method search is as follows:

$$\frac{\Gamma \vdash e \ : \ \sigma \qquad \Gamma \vdash \sigma <\!\# \, \texttt{obj}\, t.\langle\!\langle n : \tau \rangle\!\rangle \qquad \Gamma \vdash \rho <\!\# \, \sigma}{\Gamma \vdash e \hookleftarrow n \ : \ [\rho/t](t \rightarrow \tau)} \quad (search)$$

The first and the second premises assure that e is typable with a type σ that has $n : \tau$ in its protocol, while the last judgment finds a type ρ which represents exactly the type of the *self* object (whose e represents a sub-object), to which the message n was sent, and to which the body of n will be applied. Again, as in the previous (*search*) rule, there is no way of finding a type ρ matching with σ.

Expressivity. By looking at this type system and at the one of [14], we can see that there are similarities: in particular the use of the match-bound variable $u \mathrel{<\!\!\#} \texttt{obj}\, t.\langle\!\langle R \mid n : \tau \rangle\!\rangle$ in [4] plays the same role as the open object-type $\texttt{obj}\, t.r\, t$ of [14], since $r <: \lambda t.\langle\!\langle R \mid n : \tau \rangle\!\rangle$. Although this is not formally proved, we could conjecture that the expressive powers of the two systems are the same, since, by [9,1], the matching relation can be fruitfully interpreted using a higher order polymorphism.

Example 10 (Typing Example 4 in [4]). As in the type system of [14], if we want to build e' using e as a prototype, then we need to insure that p does not belong to e, and derive for the body of p the judgment:

$$u \mathrel{<\!\!\#} \texttt{obj}\, t.\langle\!\langle m : int, n : int, p : int \rangle\!\rangle \vdash \lambda self.self \Leftarrow n : [u/t](t \to int). \qquad (*)$$

Observe that the same amount of polymorphism of [14] captures the method specialization of p for all future extensions of e'. Note that the judgment $(*)$ can be used to extend e'' but not e''', since, again, e''' is not an extension of e.

3.4 Bruce's Matching-based Type System of Liquori [20]

This solution was originally adopted to add an extension operator to the Object Calculus (à la Church) of Abadi and Cardelli (cf. [20], Section 3), but can be similarly adopted to the Lambda Calculus of Objects. As in [4], this type system appeals to the notion of Matching of [8], by avoiding object subsumption over extendible objects. However, as we will show, this solution exploits a higher "degree of polymorphism" than either [14] and [4], and comparable with [15]. This will become useful when we add an explicit polymorphism in Subsection 3.7. The set of types, contexts, and judgments are exactly as in [4].

Main Typing Rules. The typing rules to build an empty object are routine; the rule to override a method, send a message to an object and search a method, instead, are the same as in [4].

The type rule to extend an object is as follows:

$$\frac{\begin{array}{l} \Gamma \vdash e_1 \ : \ \texttt{obj}\, t.\langle\!\langle R \mid \overline{m} : \overline{\sigma} \rangle\!\rangle \\ \Gamma, u \mathrel{<\!\!\#} \texttt{obj}\, t.\langle\!\langle \overline{m} : \overline{\sigma}, n : \tau \rangle\!\rangle \vdash e_2 \ : \ [u/t](t \to \tau) \quad n \text{ not in } \langle\!\langle R \mid \overline{m} : \overline{\sigma} \rangle\!\rangle \end{array}}{\Gamma \vdash \langle e_1 \leftarrow\!\!+\, n = e_2 \rangle \ : \ \texttt{obj}\, t.\langle\!\langle R \mid \overline{m} : \overline{\sigma}, n : \tau \rangle\!\rangle} \; (obj{-}ext)$$

Even if this rule looks similar to [4], the bound for u (playing the role of the type of *self*) in the judgment for e_2 has a different meaning: in fact, the $\overline{m} : \overline{\sigma}$ methods explicitly listed in the bound (and also present in the type of e_1) represent the methods that are *essential* to the typing of the body of e_2, and nothing more ([14,4] use the *entire* type of e_1 plus the method n). A consequence of this less restrictive bound is that all prototypes containing the $\overline{m} : \overline{\sigma}$ methods, and not containing the n method, can be extended with $n = e_2$, since that the type of n specializes its functionality.

Example 11 (Typing Example 4 in [20]). As in [15], in order to build the object e', using e as a prototype, we need (i) to know that p does not belongs to e, (ii) to insure that $n : int$ is present in the the type of e, and (iii) to derive for the body of p the judgment:

$$u <\!\!\# \, \mathtt{obj}\, t.\langle\!\langle n : int, p : int \rangle\!\rangle \vdash \lambda self.self \Leftarrow n : [u/t](t \to int). \qquad (*)$$

Therefore, to type the body of p, we need to know only the types of methods n (the method directly used inside p), and p (i.e. the method to be added); there is no need to extract the indirect dependencies of n (i.e. m); this is sound, since the body of p, i.e. $\lambda self.self \Leftarrow n$ only self-inflicts a message send of n to *self*. Method specialization of p will follow for all future extensions of e'. Finally, as in [15], the body of p should also be used to extend other objects such as the e'' or e''', using the same judgment $(*)$.

3.5 Binary Methods

One of the best features of the Lambda Calculus of Objects is its ability to define the so-called *binary methods* smoothly. In short, a binary method receives as input an argument of the same type of *self*. A prototypical example of such methods is the (omnipresent) equal method that has the following shape:

$$equal \triangleq \lambda self.\lambda p.(self \Leftarrow x) == (p \Leftarrow x) \&\& (self \Leftarrow y) == (p \Leftarrow y),$$

where == is the equality test among real, && is a boolean operator, and both *self* and p are points of type

$$\mathtt{obj}\, t.\langle\!\langle R \mid x : real, y : real, equal : t \to bool \rangle\!\rangle,$$

for some suitable R. Hence, in binary methods, the type-variable denoting the type of *self* (i.e. t) occurs in *contravariant* position with respect to the arrow-type constructor. As clearly stated in [7], the addition of (unrestricted) object subsumption in presence of binary methods makes the type system unsound (see also Section 6).

3.6 Comparison Between the Four Type Systems

The following table summarizes the systems with respect to the methods which are useful to build the object e' (using e as a prototype, as in Example 4) and with respect to the foundational type models.

Type System	Example 4	Type Model
[15]	$\{n, p\}$	Higher Order Rows
[14]	$\{n, m, p\}$	Higher Order Rows & Row Subtyping
[4]	$\{n, m, p\}$	Bruce's Matching
[20]	$\{n, p\}$	Bruce's Matching

The following table summarizes the typing of the $(obj{-}ext)$ rules for the body of n in all the four systems:

Type System	Judgments for e_2
[15]	$\mathit{\Gamma}, r : T \rightarrow [\overline{m}, n] \vdash e_2 \ : \ [\mathtt{obj}\, t.\langle\!\langle r\, t \mid \overline{m} : \overline{\sigma}, n : \tau \rangle\!\rangle / t](t \rightarrow \tau)$
[14]	$\mathit{\Gamma}, r <\!\!\# \, \lambda t.\langle\!\langle R \mid n : \tau \rangle\!\rangle : T \rightarrow [\,] \vdash e_2 \ : \ [\mathtt{obj}\, t.r\, t / t](t \rightarrow \tau)$
[4]	$\mathit{\Gamma}, u <\!\!\# \, \mathtt{obj}\, t.\langle\!\langle R \mid n : \tau \rangle\!\rangle \vdash e_2 \ : \ [u/t](t \rightarrow \tau)$
[20]	$\mathit{\Gamma}, u <\!\!\# \, \mathtt{obj}\, t.\langle\!\langle \overline{m} : \overline{\sigma}, n : \tau \rangle\!\rangle \vdash e_2 \ : \ [u/t](t \rightarrow \tau)$

Finally, we would like to mention that the expressive power of the systems is the same: *all partial recursive functions* are typable. We leave open the issue of decidability of the type inference problem for the four systems, and we refer to [25,26] for all related issues.

3.7 Adding Explicit Polymorphism

In our "pilgrimage" toward a Church version of the Lambda Calculus of Objects, we must pass through the addition of explicit polymorphism. To the author's knowledge, the only Church version of the Lambda Calculus of Object is the one presented in [22], based on the type system of [15]. For pedagogical reason, we prefer to introduce explicit polymorphism in a Curry setting.

In all the presented type systems, an *implicit*[8] form of polymorphism is sufficient to capture method specialization of the inherited methods. However, if we want methods to be first-class entities, then we need to add *explicit* quantification. Having explicit quantification increases the expressivity of the calculus, since it enables us to write "portable methods", i.e. methods whose bodies may not be defined at the time but provided later, i.e. resolved at run-time. We sometime refer to these functions as *mixins*. When a mixin is applied to a (pre-compiled) polymorphic method, it installs this method into the object, in a *plug-and-play* fashion. The next example clarifies the point. Consider the following function:

$$f \stackrel{\triangle}{=} \lambda p.\lambda c.\langle p \leftarrow\!+\, col = c \rangle.$$

This simple mixin cannot be typed in any of the type systems presented in this paper; the key point is that it is impossible to give a type to the object $\langle p \leftarrow\!+\, col = c \rangle$ since the body c of col as a type $selftype \rightarrow colors$ where $selftype$ denotes the type of the host object that could be an open object-type (i.e.

[8] By implicit polymorphism we mean that the universal type quantifier $\forall$ is not part of the syntax of types, but the quantified type (or row) variable is bounded (i.e. declared) in the context.

with the row-variable r occurring free) or a simple type-variable u bound in the context. Now, since c is a simple expression-variable, it should be declared *after* r [15,14] or u [4,20] and not *before*, otherwise we break the well-formation rules of contexts in all systems.

Therefore[9], if we want to type f then we must extend the set of types as follows:

$$\begin{array}{rll}
[15] & \text{Types} & \tau ::= \ldots\text{as before}\ldots \mid \forall r : \kappa.\sigma \\
[14] & \text{Types} & \tau ::= \ldots\text{as before}\ldots \mid \forall r <: R : \kappa.\tau \\
[4] & \text{Types} & \tau ::= \ldots\text{as before}\ldots \mid \forall u <\!\# \,\sigma.\tau \\
[20] & \text{Types} & \tau ::= \ldots\text{as before}\ldots \mid \forall u <\!\# \,\sigma.\tau.
\end{array}$$

The rules for polymorphic introduction/elimination and object extension becomes as follows, taking into account that, since the calculus is untyped, the operational semantics and the polymorphic abstraction and application are implicit (for related issues see also [19,30]).

In the type system of Fisher, Honsell, and Mitchell [15]

$$\frac{\Gamma, r : \kappa \vdash e : \tau}{\Gamma \vdash e : \forall r : \kappa.\tau}\ (\forall{-}I) \qquad \frac{\Gamma \vdash e : \forall r : \kappa.\tau \qquad \Gamma \vdash R : \kappa}{\Gamma \vdash e : [R/r]\tau}\ (\forall{-}E)$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 \ :\ \mathtt{obj}\, t.\langle\!\langle R \mid \overline{m} : \overline{\sigma}\rangle\!\rangle \quad \Gamma, t : T \vdash R \ :\ [\overline{m}, n] \qquad r \text{ not in } \tau \\ \Gamma \vdash e_2 \ :\ \forall r : T {\to} [\overline{m}, n].[\mathtt{obj}\, t.\langle\!\langle r\, t \mid \overline{m} : \overline{\sigma}, n : \tau\rangle\!\rangle/t](t {\to} \tau)\end{array}}{\Gamma \vdash \langle e_1 \leftarrow\!\!+\, n = e_2\rangle \ :\ \mathtt{obj}\, t.\langle\!\langle R \mid \overline{m} : \overline{\sigma}, n : \tau\rangle\!\rangle}\ (\mathit{obj}{-}\mathit{ext})$$

In the type system of Fisher [14]

$$\frac{\Gamma, r <: R : \kappa \vdash e : \tau}{\Gamma \vdash e : \forall r <: R : \kappa.\tau}\ (\forall{-}I) \qquad \frac{\Gamma \vdash e : \forall r <: R : \kappa.\tau \qquad \Gamma \vdash R' <: R}{\Gamma \vdash e : [R'/r]\tau}\ (\forall{-}E)$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 \ :\ \mathtt{obj}\, t.R \qquad \Gamma, t : T \vdash R \ :\ [n] \qquad r \text{ not in } \tau \\ \Gamma \vdash e_2 \ :\ \forall r <: \lambda t.\langle\!\langle R \mid n : \tau\rangle\!\rangle : T {\to} [\,].[\mathtt{obj}\, t.r\, t/t](t {\to} \tau)\end{array}}{\Gamma \vdash \langle e_1 \leftarrow\!\!+\, n = e_2\rangle \ :\ \mathtt{obj}\, t.\langle\!\langle R \mid n : \tau\rangle\!\rangle}\ (\mathit{obj}{-}\mathit{ext})$$

In the type system of Bono and Bugliesi [4]

$$\frac{\Gamma, u <\!\# \,\sigma \vdash e : \tau}{\Gamma \vdash e : \forall u <\!\# \,\sigma.\tau}\ (\forall{-}I) \qquad \frac{\Gamma \vdash e : \forall u <\!\# \,\sigma.\tau \qquad \Gamma \vdash \rho <\!\# \,\sigma}{\Gamma \vdash e : [\rho/u]\tau}\ (\forall{-}E)$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 \ :\ \mathtt{obj}\, t.R \qquad n \text{ not in } R \\ \Gamma \vdash e_2 \ :\ \forall u <\!\# \,\mathtt{obj}\, t.\langle\!\langle R \mid n : \tau\rangle\!\rangle.[u/t](t {\to} \tau)\end{array}}{\Gamma \vdash \langle e_1 \leftarrow\!\!+\, n = e_2\rangle \ :\ \mathtt{obj}\, t.\langle\!\langle R \mid n : \tau\rangle\!\rangle}\ (\mathit{obj}{-}\mathit{ext})$$

---

[9] **Disclaimer.** These extensions are not proved to be "type safe". However, in our modest opinion, at least the [15] one (based on a previous work [22]) should be safe.

In the type system of Liquori [20]

$$\frac{\Gamma, u <\!\# \, \sigma \vdash e : \tau}{\Gamma \vdash e : \forall u <\!\# \, \sigma.\tau} \; (\forall{-}I) \qquad \frac{\Gamma \vdash e : \forall u <\!\# \, \sigma.\tau \qquad \Gamma \vdash \rho <\!\# \, \sigma}{\Gamma \vdash e : [\rho/u]\tau} \; (\forall{-}E)$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 \;:\; \mathtt{obj}\, t.\langle\!\langle R \mid \overline{m} : \overline{\sigma} \rangle\!\rangle \qquad n \text{ not in } \langle\!\langle R \mid \overline{m} : \overline{\sigma} \rangle\!\rangle \\ \Gamma \vdash e_2 \;:\; \forall u <\!\# \, \mathtt{obj}\, t.\langle\!\langle \overline{m} : \overline{\sigma}, n : \tau \rangle\!\rangle.[u/t](t \to \tau)\end{array}}{\Gamma \vdash \langle e_1 \leftarrow\!\!\!+ n = e_2 \rangle \;:\; \mathtt{obj}\, t.\langle\!\langle R \mid \overline{m} : \overline{\sigma}, n : \tau \rangle\!\rangle} \; (obj{-}ext)$$

Remark. We have omitted the rule for method search, since these rules are the same as in the corresponding calculi with implicit polymorphism; this is not surprising, since the polymorphic row/type instantiation is performed before the self-application step.

Example 12 (Typing Example 4 revisited). Let us suppose to have a method library $\mathcal{L}$, containing a set of polymorphic method bodies to be "plugged-and-played" into e, i.e. $\mathcal{L} = \{b_p = body_p : \sigma_p, b_q = body_q : \sigma_q, \ldots\}$, with $k \geq 0$. Besides e, we want such bodies to be as polymorphic as possible, i.e. to be applicable to the biggest number of objects. We then compile $\mathcal{L}$ in the four systems. Borrowing from Example 4 the objects e', e'', and e''', we could build the following programs:

$$\begin{array}{llll} p & \triangleq \; (\lambda x.\langle e \leftarrow\!\!\!+ p = x \rangle)\, b_p & \qquad p' & \triangleq \; (\lambda x.\langle e' \longleftarrow p = x \rangle)\, b_p \\ p'' & \triangleq \; (\lambda x.\langle e'' \leftarrow\!\!\!+ p = x \rangle)\, b_p & \qquad p''' & \triangleq \; (\lambda x.\langle e''' \leftarrow\!\!\!+ p = x \rangle)\, b_p. \end{array}$$

It is not difficult to verify that, while p, p' and p'' can be type-checked in all the systems including the (already compiled) $\mathcal{L}$, p''' can be compiled only with the library compiled within the type systems of [15,20].

The following table summarizes the typing for the body of p in all systems with explicit polymorphism:

Type System	Judgments for e_2
[15]	$\Gamma \vdash e_2 \;:\; \forall r : T \to [\overline{m}, n].[\mathtt{obj}\, t.\langle\!\langle r\, t \mid \overline{m} : \overline{\sigma}, n : \tau \rangle\!\rangle / t](t \to \tau)$
[14]	$\Gamma \vdash e_2 \;:\; \forall r <: \lambda t.\langle\!\langle R \mid n : \tau \rangle\!\rangle : T \to [\;].[\mathtt{obj}\, t.r\, t/t](t \to \tau)$
[4]	$\Gamma \vdash e_2 \;:\; \forall u <\!\# \, \mathtt{obj}\, t.\langle\!\langle R \mid n : \tau \rangle\!\rangle.[u/t](t \to \tau)$
[20]	$\Gamma \vdash e_2 \;:\; \forall u <\!\# \, \mathtt{obj}\, t.\langle\!\langle \overline{m} : \overline{\sigma}, n : \tau \rangle\!\rangle.[u/t](t \to \tau)$

4 Curry versus Church

In this section we try to transform the Lambda Calculus of Objects à la Curry into a corresponding calculus à la Church. To do this, we decorate it with types. The goals we want to achieve in this transformation are as follows:

- *uniqueness of typing* (mandatory); intuitively, this property says that we could transform a type system into an equivalent one where for all expressions of our calculus, there is *at most* one type rule to apply (those systems

are usually called *algorithmic*). Since we do not deal with object subsumption, it follows that every expression has an unique type.
- *explicit polymorphism and first-class method bodies* (mandatory); in short, without explicit polymorphism the language cannot be "fully typed" in its total sense, since there will be expressions whose polymorphism is not explicitly annotated in the expressions itself. As we have seen in Subsection 3.7, adding explicit polymorphism enable us to build mixins.
- *binary methods and/or static typing*; these two requirements are strictly related, since very often the possibility to have binary methods leads to a type-driven semantics (mandatory if you need *multiple dispatching*). For this and related issues, see the seminal paper of [11], and also [7,12,22]. As we will see, by avoiding subtyping, we will keep binary methods and static type checking.

For obvious lack of space, we therefore cannot apply all systems to the Lambda Calculus of Objects à la Church. We choose to adopt the type system of [14], even if it is "less polymorphic" than the one of [15,20], since it allows a nice and smooth extension into a calculus à la Church.

4.1 The Lambda Calculus of Objects à la Church

The set of types is defined as in Subsection 3.7. The expressions of the fully decorated calculus are defined by the following grammar:

$$
\begin{array}{lll}
e ::= c \mid x \mid \lambda x : \sigma . e \mid e_1\, e_2 \mid & & \text{(typed } \lambda\text{-calc.)} \\
\quad\;\; e\, R \mid \lambda r <: R : \kappa . e \mid & & \text{(polymorphic expr.)} \\
\quad\;\; \langle\, \rangle \mid e \Leftarrow m \mid \langle e_1 \longleftarrow m = e_2 \rangle \mid \langle e_1 \leftarrow\!\!\!+\, m = e_2 \rangle \mid & & \text{(object expr.)} \\
\quad\;\; e \hookleftarrow m, & & \text{(auxiliary expr.)}
\end{array}
$$

where the last two forms represent the polymorphic lambda abstraction and application.

Small-step Reduction Semantics. The small-step operational semantics is type-driven: its core is given by the following reduction rules:

$$
\begin{array}{llll}
(\mathit{Beta}) & (\lambda x : \sigma . e_1)\, e_2 & \stackrel{ev}{\to} [e_2/x]e_1 & \\
(\mathit{Select}) & e \Leftarrow m & \stackrel{ev}{\to} (e \hookleftarrow m)\, (\lambda t . R)\, e & \text{if } \Gamma \vdash_{\mathcal{A}} e : \mathtt{obj}\, t.R^{10} \\
(\mathit{Succ}) & \langle e_1 \leftarrow\!\!* \; m = e_2 \rangle \hookleftarrow m & \stackrel{ev}{\to} e_2 & \\
(\mathit{Next}) & \langle e_1 \leftarrow\!\!* \; n = e_2 \rangle \hookleftarrow m & \stackrel{ev}{\to} e_1 \hookleftarrow m. &
\end{array}
$$

As for the calculus à la Curry with explicit polymorphism, the body of a method is a (explicitly annotated) polymorphic lambda abstraction; it follows that before

[10] To be precise, $\lambda t.R$ becomes $\lambda t : T.R$. We could also add the stronger condition $m \in R$ (i.e. $R \equiv \langle\!\langle R' \mid m : \tau \rangle\!\rangle$, for some R' and τ), but this constraint is guaranteed when $(e \hookleftarrow m)$ is well-typed.

applying the body of the method to *self*, we need to instantiate the polymorphic lambda abstraction correctly with an adequate row-abstraction. To deduce this row, we use an algorithmic type system, denoted by $\vdash_{\mathcal{A}}$, equivalent to the one of [14], obtained by using standard techniques[11].

Big-step Operational Semantics. We extend the set of results as follows:

$$v ::= \ldots \text{as before} \ldots \mid \lambda r <: R : \kappa . e.$$

The big step semantics is equally type-driven, but the only point where a type derivation is needed is the $(Red{-}Select)$ rule:

$$\frac{\begin{array}{l}\varepsilon \vdash_{\mathcal{A}} e : \texttt{obj}\, t.R^{12} \\ e \Downarrow obj \qquad obj \hookleftarrow m \Downarrow \lambda r <: R' : \kappa . e' \qquad ([(\lambda t.R)/r]e')\, e \Downarrow v\end{array}}{e \Leftarrow m \Downarrow v} \quad (Red{-}Select)$$

In this rule, before the self-application, we need to instantiate correctly the polymorphic body of n with a suitable row-abstraction (i.e. R) obtained invocating the algorithmic type system $\vdash_{\mathcal{A}}$ on e.

4.2 The Type System à la Church Inspired to [14]

We will not present the full set of rules in detail: the rules of explicit introduction of polymorphism are similar to those presented in Subsection 3.7, taking into account that here term decoration is explicit:

$$\frac{\Gamma, r <: R : \kappa \vdash_{\mathcal{A}} e : \tau}{\Gamma \vdash_{\mathcal{A}} \lambda r <: R : \kappa . e : \forall r <: R : \kappa . \tau} (\forall{-}I) \qquad \frac{\Gamma \vdash_{\mathcal{A}} e : \forall r <: R : \kappa . \tau \quad \Gamma \vdash_{\mathcal{A}} R' <: R}{\Gamma \vdash_{\mathcal{A}} e\, R' : [R'/r]\tau} (\forall{-}E)$$

The only rule that merits to be mentioned is the one concerning method search in the algorithmic type system. It behaves as follows:

$$\frac{\Gamma \vdash_{\mathcal{A}} e \ : \ \texttt{obj}\, t.\langle\!\langle R \mid n : \tau \rangle\!\rangle}{\Gamma \vdash_{\mathcal{A}} e \hookleftarrow n \ : \ \forall r <: (\lambda t.\langle\!\langle R \mid n : \tau \rangle\!\rangle) : T \to [\,].[\texttt{obj}\, t.r\, t/t](t \to \tau)} \quad (search)$$

This rule simply says that a body of a method is a polymorphic lambda abstraction, bounded by $\lambda t.\langle\!\langle R \mid n : \tau \rangle\!\rangle$, since $\texttt{obj}\, t.\langle\!\langle R \mid n : \tau \rangle\!\rangle$ is the type of the object e in which we are performing the recursive (right-to-left) traversing in order to reach the right-most definition of n.

[11] Intuitively, we usually remove all the subrowing rules concerning transitivity and reflexivity, and we rearrange the rules for expressions to take these modifications into account (see [12]).

[12] In this case, we could also add the judgment $t : T \vdash_{\mathcal{A}} R <: R'$, but the subrowing condition is enforced by the well-typedness of $obj \hookleftarrow m$.

Exercise. If you'd like to play with explicit polymorphism and Church calculi, then you should first try to adapt the operational semantics (small and big-step) to the remaining type systems, and then write the most interesting type rules concerning polymorphism and method extraction.

4.3 Binary Methods

Also in the Lambda Calculus of Object à la Church, binary methods are allowed. The equal method of Section 3.5 is rewritten à la Church with [14] types as:

$$\begin{aligned} \mathit{equal} \;\triangleq\; & \lambda r <: (\lambda t.\langle\!\langle R \mid \mathit{equal} : t \to \mathit{bool} \rangle\!\rangle : T \to [\,]).\lambda \mathit{self} : \texttt{obj}\, t.r\, t. \\ & \lambda p : \texttt{obj}\, t.r\, t.(\mathit{self} \Leftarrow x) == (p \Leftarrow x) \&\& (\mathit{self} \Leftarrow y) == (p \Leftarrow y), \end{aligned}$$

of type $\forall r <: (\lambda t.\langle\!\langle R \mid \mathit{equal} : t \to \mathit{bool} \rangle\!\rangle : T \to [\,]).\, [\texttt{obj}\, t.r\, t/t](t \to t \to \mathit{bool})$, for some suitable R representing the methods of the prototype e extended with the *equal* method.

4.4 Switching to an Untyped Operational Semantics

All type systems presented here were conceived to work without the subsumption rule (*subsume*) presented at the very beginning of Section 3. The rest of this section is devoted to explain how a type-driven operational semantics can be switched into an untyped one in a calculus without subsumption.

Following [22], we could build an *erasing* function $\mathcal{E}$ from typed to untyped expressions, so that every expression derivable à la Curry (with explicit polymorphic types) can be derivable as the erasure of some typed expression à la Church. The erasing function $\mathcal{E}$ simply erases the type annotations on the abstracted variables, and eliminates the expression-row applications:

$$\begin{array}{rclrcl} \mathcal{E}(e\,R) &=& \mathcal{E}(e) & \mathcal{E}(\lambda x : \tau.e) &=& \lambda x.\mathcal{E}(e) \\ \mathcal{E}(\lambda r <: R : \kappa.e) &=& \mathcal{E}(e) & \mathcal{E}(e \hookleftarrow n) &=& \mathcal{E}(e) \hookleftarrow n \\ \mathcal{E}(\forall r <: R : \kappa.\tau) &=& \mathcal{E}(\tau) & \mathcal{E}(\langle\!\langle R \mid n{:}\tau \rangle\!\rangle) &=& \langle\!\langle \mathcal{E}(R) \mid n{:}\mathcal{E}(\tau) \rangle\!\rangle \\ \mathcal{E}(\lambda t : T.R) &=& \lambda t.\mathcal{E}(R) & & & \end{array}$$

In all the other expressions, $\mathcal{E}$ is compositional or the identity.

It is not difficult to see that the type-driven reduction semantics becomes the untyped one of Section 2.

The following lemma ensures that, after type checking, the computation of an expression of the typed calculus can be performed on its erasure, since the result will be the same (modulo erasure).

Lemma 13. *Let $\Gamma \vdash e : \tau$ in the Lambda Calculus of Objects à la Church. If $\mathcal{E}(e) \overset{ev}{\twoheadrightarrow}_u e'$, then there exists e'' such that $e' \equiv \mathcal{E}(e'')$ and $e \overset{ev}{\twoheadrightarrow}_t e''$, where $\overset{ev}{\twoheadrightarrow}_u$ and $\overset{ev}{\twoheadrightarrow}_t$ denotes the untyped and the type-driven small-step operational semantics, respectively.*

5 A Calculus based on [2]

As we said in Section 2, in our Lambda Calculus of Objects, the objects are considered as lists of pairs *(method names-method bodies)*, and method lookup is performed via a right-to-left traversing of the object until the searched method is found.

In the Object Calculus of [2], instead, objects are considered as *sets* of pairs *(method names-method bodies)*. Although it is well-known that a set can be implemented with a list, this view of objects has some interesting properties:

- it enables us to build an operational semantics where object override (and from [21] also object extension) could be axiomatized via a simple rewriting step that reduces a more complex object, usually an object where some methods are redefined, into a simpler one, where the older methods are substituted with the newly redefined ones.
- it enables us to perform method lookup *in one single step*, since the lookup is reduced to a *simple set membership test*, thus avoiding all matters of typability of explicit method lookup.

For reason of simplicity, in this section we deal only with calculi à la Curry. We introduce a calculus inspired to the Object Calculus, where objects are sets, as follows:

$$e ::= x \mid \lambda x.e \mid e_1 e_2 \mid \langle \overline{m} = \overline{e} \rangle \mid e \leftarrow\!\!+\, m = e' \mid e \longleftarrow m = e' \mid e \Leftarrow m.$$

The $\leftarrow\!\!+$, and $\longleftarrow$ (shortly $\longleftarrow\!\!*$) operators have the following (informal) signature: $\longleftarrow\!\!*$ :: (Sets × Method Names × Method Bodies) ⇒ Sets. Observe that, in the object expressions, the order of methods is unimportant.

There are some differences between this calculus and the Object Calculus:

- the Object Calculus does *not* include all the expressions related to the lambda calculus that are, instead, codified into suitable objects;
- in the Object Calculus, the body of methods always has the shape $\varsigma(self)e$, where ς is a binder (as λ) that occurs only *inside* objects, and e is an object expression. Here, instead, a body of a method can be any expression which can reduce to a lambda abstraction.

In this section, we do not deal with types, since all four type systems presented in this paper could be easily adapted with minor changes.

5.1 Operational Semantics

Small-step Operational Semantics. The small-step operational semantics is directly inspired to [2] and it is axiomatized by the following rewriting steps:

$$\begin{array}{llll}
(Select) & \langle \overline{m} = \overline{e}, n = e \rangle \Leftarrow n & \stackrel{ev}{\to} e\ \langle \overline{m} = \overline{e}, n = e \rangle & \\
(Override) & \langle \overline{m} = \overline{e}, n = e' \rangle \longleftarrow n = e & \stackrel{ev}{\to} \langle \overline{m} = \overline{e}, n = e \rangle & \\
(Extension) & \langle \overline{m} = \overline{e} \rangle \leftarrow\!\!+\, n = e & \stackrel{ev}{\to} \langle \overline{m} = \overline{e}, n = e \rangle & n \notin \overline{m} \\
(Beta) & (\lambda x.e_1)e_2 & \stackrel{ev}{\to} [e_2/x]e_1. &
\end{array}$$

Note that in [2] the (*Beta*) axiom can be simulated with the help of the rules (*Select*) and (*Override*) rules.

Big Step Operational Semantics. Building a big-step semantics is quite simple. It is essentially what we have shown in Section 2, except that now the dynamic method lookup is implicit; this means that we need to reconsider the (*Red−Select*) rule, and add two rules for overriding and extending objects. The semantics uses the (*Red−Val*) and (*Red−Beta*) deduction rules of Section 2, plus the following rules:

$$\frac{e \Downarrow \langle \overline{m} = \overline{e}, n = e' \rangle \qquad e' \Downarrow \lambda x.e'' \qquad [\langle \overline{m} = \overline{e}, n = e' \rangle / x]e'' \Downarrow v}{e \Leftarrow n \Downarrow v} \quad (\mathit{Red{-}Select})$$

$$\frac{e \Downarrow \langle \overline{m} = \overline{e}, n = e'' \rangle}{e \longleftarrow n = e' \Downarrow \langle \overline{m} = \overline{e}, n = e' \rangle} \quad (\mathit{Red{-}Over})$$

$$\frac{e \Downarrow \langle \overline{m} = \overline{e} \rangle \qquad n \notin \overline{m}}{e \leftarrow\!\!+\, n = e' \Downarrow \langle \overline{m} = \overline{e}, n = e' \rangle} \quad (\mathit{Red{-}Ext})$$

Also in this case, the (*Red−Beta*) deduction rule can be simulated with the help of the remaining deduction rules.

One can easily see that the (*Red−Select*) rule of Section 2 and the above (*Red−Select*) rule are very similar. This is not surprising: in fact, in the former rule the method lookup phase is explicit with the help of the (*Red−Next*) and (*Red−Succ*) rules, while in the latter the method lookup phase is hidden in the set notation. This approach greatly simplify all the proofs; in particular, the one of subject reduction.

Moreover, as we can observe in the above (*Red−Select*) rule, the body e' of the method n reduces to a lambda abstraction $\lambda x.e''$. This abstraction will be applied directly (as in the (*Red−Beta*) rule) by substituting for every occurrences of x in the body of e' the object itself: this, very informally, means to *open* first the encapsulated method n and then to *evaluate* the body of e' of n.

6 About Object Subsumption

Object subsumption has been fundamental in the object-oriented paradigm, since it allows a significant reuse of code. Unfortunately, (see [16,2]), extension in presence of subtyping makes the type system unsound. In this final section, we try to point out the solutions (without criticism) proposed in the literature which add object subsumption to object calculi with an extension operator.

- one may permit unrestricted subsumption in absence of binary methods, maintaining static type checking, by allowing occurrences of the type of *self* only in covariant position (cf, among others, [2,17,20,28]). We can also use the *variance annotations* of [2]

– one may decide to keep binary methods in presence of subsumption, and use a type-driven operational semantics in the style of Castagna *et alt* [13,11,12];
– one may restrict subtyping by "hiding" only the methods that are not referred to any other method [6,5];
– one may not forget the type of the "hidden methods" in order to guarantee that any future addition of those methods will have a type consistent with the hidden one [21,20];
– one may use method dictionaries inside object-types [29], and permitting to re-add an "hidden" method with two different types.

A detailed comparison of calculi with object extension in presence of object subsumption is under development.

7 Conclusions

We hope to have contributed to the understanding of some basic concepts of object calculi and other related type systems which have been presented in the literature. Object-based calculi are not as well-know as the class-based ones, but their importance will increase especially when mixed with class-based features, such as in the *Beta* [23], or *O*2 of [2].

Acknowledgments. I would like to thank the *Logics of Programs* group of the Department of Computer Science of Turin, for the time spent to teach me the fundamentals of Theoretical Computer Science, to Furio Honsell for his fruitful discussions and suggestions, Stefania Garlatti-Costa for the careful reading of the paper and one anonymous referee for the insightful comments which helped me to improve the paper greatly.

References

1. M. Abadi and L. Cardelli. On Subtyping and Matching. In *Proc. of ECOOP*, volume 952 of *Lecture Notes in Computer Science*, pages 145–167. Springer–Verlag, 1995.
2. M. Abadi and L. Cardelli. *A Theory of Objects.* Springer-Verlag, 1996.
3. G. Bellè. Some Remarks on Lambda Calculus of Objects. Technical report, Dipartimento di Matematica ed Informatica, Università di Udine, 1994.
4. V. Bono and M. Bugliesi. Matching Constraints for the Lambda Calculus of Objects. In *Proc. of TLCA*, volume 1210 of *Lecture Notes in Computer Science*, pages 46–62. Springer-Verlag, 1997.
5. V. Bono, M. Bugliesi, M. Dezani-Ciancaglini, and L. Liquori. Subtyping Constraint for Incomplete Objects. In *Proc. of TAPSOFT/CAAP*, volume 1214 of *Lecture Notes in Computer Science*, pages 465–477. Springer-Verlag, 1997.
6. V. Bono and L. Liquori. A Subtyping for the Fisher-Honsell-Mitchell Lambda Calculus of Objects. In *Proc. of CSL*, volume 933 of *Lecture Notes in Computer Science*, pages 16–30. Springer-Verlag, 1995.
7. K. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G. Leavens, and B. Pierce. On Binary Methods. *Theory and Practice of Object Systems*, 1(3), 1996.
8. K.B. Bruce. A Paradigmatic Object–Oriented Programming Language: Design, Static Typing and Semantics. *Journal of Functional Programming*, 4(2):127–206, 1994.

9. K.B. Bruce, A. Shuett, and R. van Gent. Polytoil: a Type-safe Polymorphic Object Oriented Language. In *Proc. of ECOOP*, volume 952 of *Lecture Notes in Computer Science*, pages 16–30, 1995.
10. L. Cardelli. A Language with Distributed Scope. *Computing System*, 8(1):27–59, 1995.
11. G. Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, 1995.
12. G. Castagna. *Object-Oriented Programming: A Unified Foundation*. Progress in Theoretical Computer Science. Birkäuser, Boston, 1996.
13. G. Castagna, G. Ghelli, and G. Longo. A Calculus for Overloaded Functions with Subtyping. *Information and Computation*, 117(1):115–135, 1995.
14. K. Fisher. *Type System for Object-Oriented Programming Languages*. PhD thesis, University of Stanford, August 1996.
15. 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.
16. K. Fisher and J. C. Michell. Notes on Typed Object-Oriented Programming. In *Proc. of TACS*, volume 789 of *Lecture Notes in Computer Science*, pages 844–885. Springer-Verlag, 1994.
17. K. Fisher and J. C. Mitchell. A Delegation-based Object Calculus with Subtyping. In *Proc. of FCT*, volume 965 of *Lecture Notes in Computer Science*, pages 42–61. Springer-Verlag, 1995.
18. A. Goldberg and D. Robson. *Smalltalk-80: the Language and its Implementation*. Addison-Wesley, 1983.
19. D. Leivant. Polymorphic Type Inference. In *Proc. of the 10th ACM Symposium on Principles of Programming Languages*, pages 88–98. The ACM Press, 1983.
20. L. Liquori. Bounded Polymorphism for Extensible Objects. Technical Report CS-24-96, Computer Science Department, University of Turin, Italy, 1996.
21. L. Liquori. An Extended Theory of Primitive Objects: First Order System. In *Proc. of ECOOP*, volume 1241 of *Lecture Notes in Computer Science*, pages 146–169. Springer-Verlag, 1997.
22. L. Liquori and G. Castagna. A Typed Lambda Calculus of Objects. In *Proc. of Asian*, volume 1179 of *Lecture Notes in Computer Science*, pages 129–141. Springer-Verlag, 1996.
23. O.L. Madsen, Moller-Pedersen, and K. B. Nygaard. *Object-Oriented programming in the Beta programming language*. Addison-Wesley, 1993.
24. P. Paladin. Teoremi di Congruenza per Lambda-Calcoli Orientati agli Oggetti. Master's thesis, Dipartimento di Matematica ed Informatica, Università di Udine, 1993. In Italian.
25. J. Palsberg. Efficient Inference of Object Types. In *Proc. of LICS*, pages 186–195, 1993.
26. J. Palsberg and T. Jim. Type Inference for Simple Object Types is NP-Complete. *Nordic Journal of Computing*, 1997.
27. Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Denmark, 1981.
28. D. Rémy. From Classes to Objects via Subtyping. In *Proc. of ESOP*, 1998.
29. J.G. Riecke and C. Stone. Privacy via Subsumption. In *Electronic proceedings of FOOL-98*, 1998.
30. S. van Bakel, L. Liquori, S. Ronchi della Rocca, and P. Urzyczyn. Comparing Cubes of Typed and Type Assignment System. *Annals of Pure and Applied Logics*, 86(3):267–303, 1997.