Subtyping Constraints for Incomplete Objects

(Extended Abstract)

Viviana Bono*, Michele Bugliesi**
Mariangiola Dezani–Ciancaglini*, Luigi Liquori*

* Dip. Informatica, Università di Torino, C.so Svizzera 185, 10149 Torino, Italy
** Dip. Matematica, Università di Padova, Via Belzoni 7, 35131 Padova, Italy

Abstract. We extend the type system for the *Lambda Calculus of Objects* [14] to account for a notion of *width* subtyping. The main novelties over previous work are the use of bounded quantification to achieve a new and more direct rendering of *MyType* polymorphism, and a uniform treatment for other features that were accounted for via different systems in subsequent extensions [7, 6] of [14]. In particular, the new system provides for (*i*) appropriate type specialization of inherited methods, (*ii*) static detection of errors, (*iii*) *width* subtyping compatible with object extension, and (*iv*) complete freedom in the order of method addition.

1 Introduction

In the last ten years, many theoretical studies have addressed the problem of deriving safe and flexible type systems for object-oriented languages. The interest of these studies has initially been centered around *class-based* languages like *Smalltalk* [16], and has subsequently been directed to *delegation-based* languages, such as *Self* [21]. Despite the conceptual differences between the underlying object-oriented models[1], several ideas originated from the experience on class-based languages have proved useful in the development of type systems for delegation-based languages. For instance, the notion of *row-variable* introduced by [22] to type extensible records was refined in [14, 7, 15, 6, 20] to type extensible objects. Similarly, the *recursive record-types*, introduced to provide functional models of class-based languages [11, 9, 13, 12], have then been applied to characterize object calculi supporting method override in presence of object-subsumption [3].

A further important notion that originated in the study of class-based models, (as well as in the record calculus of [10]) is that of (*F*-)*bounded quantification* as a tool for modeling the subclass relation. Unlike other notions, to our knowledge, the role of bounded quantification has not been as yet investigated in the context of delegation-based languages, where method extension occurs at the object-level rather than at the class level[2]: this paper makes a first step in this direction.

[1] Briefly: in class-based languages, objects are created by class instantiation and inheritance takes place at the class level. In delegation-based models, instead, objects are created from existing objects used as prototypes, and inheritance occurs at the object-level.

[2] The higher-order system of the Object Calculus [3] does, in fact, use bounded quantification to capture a notion of method extension. However, in this calculus extension is only allowed on classes, not on objects.

The *Lambda Calculus of Objects* is an untyped λ-calculus enriched with object forms and three primitive operations on objects: *method addition*, to define new methods, *method override*, to redefine methods, and *method call*, to send a message to (i.e., invoke a method on) an object. In [14] a type system for this calculus is defined, that provides for static detection of errors, such as *message not understood*, while at the same time allowing types of methods to be specialized to the type of the inheriting objects. This mechanism, that is commonly referred to as *MyType* specialization, is rendered in the type system in terms of a form of higher-order polymorphism which, in turn, uses implicit quantification over *row-schemes* to capture the underling notion of *protocol extension*.

The type system we present in this paper develops on the original work of [14] and subsequent extensions [7, 6]. We next briefly review these proposals and discuss the relations with our present approach.

In [7], an extension of the system of [14] is presented that gives provision for subtyping. The subtype relation arises from using *labeled-types* to allow methods to be "hidden" from the type of an object, subject to the constraint that "hidden" methods are not referenced to by other methods in the type.

In [6], an orthogonal extension of [14] is proposed that allows objects to be typed independently of the order of their method addition[3]. This flexibility arises in [6] from introducing the notion of *completion*, a complement to *interface*, to convey information on (the types of) methods that are not available from the object, and yet are referenced to by the methods of the interface. Besides allowing a more flexible typing of methods (in particular, of mutually recursive method definitions), this extension also gives provision for method invocation when the receiver of the message is an *incomplete* object, i.e. an object whose implementation (i.e. the set of its methods) is only partially specified.

The approach we take in this paper combines the mechanism for subtyping proposed in [7] with a support for incomplete objects, peculiar also to [6], that allows prototypes to be defined, and operated with as well, while part of their implementation (i.e. their methods) is yet to be defined. As a result, the present type system supports the following features:

- appropriate method specialization of inherited methods;
- static detection of errors, such as *message-not-understood*;
- "width" subtyping compatible with method extension;
- complete freedom in the order of method addition.

The main novelties over previous work are a uniform treatment for the features that were accounted via different systems in [14, 7, 6], and a new and more direct rendering of *MyType* polymorphism for method bodies.

In [14, 7, 6], type specialization is captured by introducing notions such as *row-variables*, *higher-order rows* and *row-application* which, in turn, require a rule of β–reduction in the calculus of rows and types. Here, instead, method specialization is rendered directly in terms of subtyping and (implicit) bounded

[3] In [14] the addition of an m method to an object can be typed only if all the methods that are referenced to (via message send or method override) in the body of m are already available from that object.

quantification. Technically, the new solution is based on allowing in our contexts occurrences of type-variables that are subtypes of suitable class–types (i.e. types of objects). These type-variables, which are implicitly universally quantified, are used within the types of methods to build methods as polymorphic functions. The subtyping constraints are then used to enforce correct instantiations of the method types as these methods get inherited.

Although the original system appears superior to the present one in terms of a possible encoding in *LF* [17], the new system does have the advantage of reducing the technical overhead of the calculus of rows and types in [14] and subsequent extensions, and hence to allow a simpler and more direct proof of Subject Reduction.

The rest of the paper is organized as follows. In Section 2, we briefly overview the untyped calculus of [14] with the operational semantics of [7]. In Section 3, we present the new typing rules for objects. Some motivating examples are presented in Section 4, while, in Section 5, we prove type soundness. Finally, we conclude in Section 6 with some additional remarks, and a discussion on related papers.

2 The Untyped Calculus

An expression of the untyped calculus can be any of the following:

$$e \ ::= \ x \mid c \mid \lambda x.e \mid e_1\, e_2 \mid \langle\rangle \mid e \Leftarrow m \mid \langle e_1 \hookleftarrow m{=}e_2\rangle \mid e \hookleftarrow m \mid err,$$

where x is a variable, c a constant and m is a method name. The reading of the object-related forms is as follows:

$\langle\rangle$ is the empty object;

$e \Leftarrow m$ sends message m to object e;

$e \hookleftarrow m$ searches the body of the m method within object e;

$\langle e_1 \hookleftarrow m{=}e_2\rangle$ extends object e_1 with a method m having body e_2;

err represents run-time errors.

As in [14], the expression $\langle e_1 \hookleftarrow m{=}e_2\rangle$ is typeable only when e_1 has a type whose interface does *not* contain the m method; the difference, here, is that methods may be added to the same object more than once, provided that they are ***hidden*** from the interface of the object-type prior to a new addition. Note also that method addition is the only operator available for modifying the structure of an object: as we shall see, the rules for subtyping allow a uniform treatment of the operations of addition and override (that were instead distinguished in [14, 7, 15, 6]) without affecting static typing (see Section 3.2).

Besides method addition, the other main operation on objects is method invocation, whose intuitive semantics may be stated as follows: when an object e containing an m method is sent the message m, the result is obtained by applying the body of m to the object e itself. In defining the operational semantics of the calculus, we must therefore give, besides the rules of β-reduction and method invocation, also a mechanism for extracting the appropriate method out of an object. As suggested in [14], a natural way to approach this is to use a permutation rule like the following:

$$\langle\langle e \hookleftarrow m{=}e_1\rangle \hookleftarrow n{=}e_2\rangle \ = \ \langle\langle e \hookleftarrow n{=}e_2\rangle \hookleftarrow m{=}e_1\rangle,$$

whenever m and n are distinct method names. Given this equational rule, the semantics of method invocation would then be stated simply as a reduction from the message sent $\langle e_1 \leftarrow\!\circ\, m{=}e_2\rangle \Leftarrow m$ to the application $e_2\, \langle e_1 \leftarrow\!\circ\, m{=}e_2\rangle$. In [6], it is shown that this form of method permutation can be soundly accounted in a system without subtyping. Unfortunately, however, in the presence of subtyping, permuting the order of two method additions within an object may change the type of the object, thus making the above equation unsound.

Therefore, in the definition of the operational semantics, we adopt a different solution that uses the search operator '$\hookleftarrow$' to inspect the structure of objects and perform method extraction. The core of the operational semantics is given by the following reduction rules:

(β)	$(\lambda x.e_1)\, e_2$	$\rightarrow$	$[e_2/x]\, e_1$
$(\Leftarrow)$	$e \Leftarrow m$	$\rightarrow$	$(e \hookleftarrow m)\, e$
$(\hookleftarrow\ succ)$	$\langle e_1 \leftarrow\!\circ\, m{=}e_2\rangle \hookleftarrow m$	$\rightarrow$	e_2
$(\hookleftarrow\ next)$	$\langle e_1 \leftarrow\!\circ\, n{=}e_2\rangle \hookleftarrow m$	$\rightarrow$	$e_1 \hookleftarrow m.$

The rule (β) is standard, while the remaining rules formalize the semantics of method invocation as the result of *search* and *self-application*: evaluating $e \Leftarrow m$ leads to evaluating the application $(e \hookleftarrow m)\, e$, where $e \hookleftarrow m$ returns the body of the m method that is then applied to e itself. Method search is performed by a recursive traversal of the "sub-objects" of e that succeeds upon reaching the right-most addition of the method in question. The use of search expressions in our calculus is inspired to [7], and it provides a more direct technical device than the *bookkeping* relation originally introduced in [14]. Type soundness for this extraction mechanism is a direct consequence of subject reduction, while it required the definition of an evaluation strategy with mutually–recursive functions in [14, 6].
The reduction relation includes additional rules (given in Table 1) that capture "incorrect" computations leading to run-time errors. The operational semantics $\xrightarrow{eval}\!\!\!\!\rightarrow$ is then defined as the reflexive, transitive and contextual closure of the reduction relation.

$(fail\ \langle\rangle)$	$\langle\rangle \hookleftarrow n \xrightarrow{eval} err$	$(fail\ abs)$	$\lambda x.e \hookleftarrow n \xrightarrow{eval} err$
$(err\ appl)$	$err\ e \xrightarrow{eval} err$	$(err \hookleftarrow)$	$err \hookleftarrow n \xrightarrow{eval} err$

Table 1. Rules for *err*.

3 Static Type System

The type system gives provision for incomplete objects in ways similar to [6]. Incomplete objects behave operationally as "standard" objects whose methods may be invoked via corresponding messages. Their typing, instead, is different, in that an incomplete object may be typed even though it contains references (via message sends or extensions) to methods that are yet to be added. The type of an incomplete object is defined by a class-type expression of the form:

$$\mathtt{class}\, t.\langle\!\langle m_1{:}\alpha_1, \ldots, m_k{:}\alpha_k\rangle\!\rangle \circ \langle\!\langle p_1{:}\gamma_1, \ldots, p_l{:}\gamma_l\rangle\!\rangle,$$

where the m_i's and p_i's are method names, whereas the α_i's and the γ_i's are *labeled* types (whose role is discussed below). Given the above class-type, we refer to the two components $\langle\!\langle m_1{:}\alpha_1, \ldots, m_k{:}\alpha_k \rangle\!\rangle$ and $\langle\!\langle p_1{:}\gamma_1, \ldots, p_l{:}\gamma_l \rangle\!\rangle$ as, respectively, the *interface–* and *completion–rows* of the type. The binder `class` scopes over the two rows, and the bound variable t may occur free within the scope of the binder, with every free occurrence referring to the class–type itself.

The interface–row of a class–type describes all the methods (and their types) that have been added to the objects of that type. The completion-row, instead, conveys approximate information on (the types of) the methods that have not been added to an object, and yet are referenced to by the methods already available from that object. Thus, intuitively, methods listed in the completion-row of a class-type are those methods that are needed to "complete" objects with that type. The ability to give a type to an object, even though it is incomplete, derives from the use of labeled-types. Labeled-types bear essentially the same meaning as in [7]: if $\alpha \equiv \tau_\Delta$ is the labeled-type of an m method within an object, then Δ provides the names of the remaining methods of that object upon which m may depend. Unlike [7], labels contain also indirect dependencies, i.e. the label Δ contains the names of the methods referenced to by m in a send or an override for *self*, together with the methods referenced to by these methods and so on. This encoding still allows new types to be derived, by subtyping, from a given class-type: the new types arise from *hiding*, from the rows of the given type, (types of) methods that do not occur in the labeled-types of the methods which remain in the interface-row.

3.1 Types and Rows

The type expressions include type-constants, type-variables, function-types and class-types. The symbol ι denotes type-constants, t, u, and v denote type-variables, σ, τ, ρ, denote types, whereas α, β, γ, denote labeled-types. All symbols may appear indexed by integers.

The set of labels, rows, and types are defined inductively as follows:

Labels	$\Delta ::= \{m_1, \ldots, m_k\}$	$(k \geq 0)$
Rows	$R ::= \langle\!\langle\rangle\!\rangle \mid \langle\!\langle R \mid m{:}\tau_\Delta \rangle\!\rangle$	with $m \notin \mathcal{M}(R), m \notin \Delta$
Types	$\tau ::= \iota \mid t \mid \tau \rightarrow \tau \mid \texttt{class}\, t.R_1 \circ R_2$	with $\mathcal{M}(R_1) \cap \mathcal{M}(R_2) = \{\}$

where the set $\mathcal{M}(R)$ of method names of a row R is inductively defined by:

$$\mathcal{M}(\langle\!\langle\rangle\!\rangle) = \{\}, \quad \text{and} \quad \mathcal{M}(\langle\!\langle R \mid m{:}\tau_\Delta \rangle\!\rangle) = \mathcal{M}(R) \cup \{m\}.$$

Row expressions that differ only for the order of $m{:}\alpha$ pairs, or for the name of the type-variable bound by `class` are considered identical.

Although the interface- and completion-rows of a class–type are structurally equivalent, we will often find it convenient to distinguish their role by choosing different labels, namely, R and C stand for arbitrary interface–rows and completion–rows, respectively.

As an important remark, we note that, in contrast to the systems of [14, 7, 6], our types are defined independently from row-variables, higher-order rows, applications of rows to types, and kinds. This allows a simplification over these proposals as, having no β-reduction for types, our type derivations are in normal-form by construction.

The contexts are defined a follows: $\Gamma ::= \varepsilon \mid \Gamma,\, x : \tau \mid \Gamma,\, u \preceq \tau$. Judgments have the form $\Gamma \vdash *$, $\Gamma \vdash e : \tau$, and $\Gamma \vdash \tau_1 \preceq \tau_2$, where $\Gamma \vdash *$ can be read as "Γ is a well-formed context" and the meaning of the other judgments is the usual one. Table 2 shows the formation rules for contexts.

$$\frac{}{\varepsilon \vdash *}\ (\mathit{start}) \qquad \frac{\Gamma \vdash * \quad x \notin \Gamma}{\Gamma,\, x{:}\tau \vdash *}\ (\mathit{var})$$

$$\frac{\Gamma \vdash * \quad u \notin \Gamma \quad u \notin \tau}{\Gamma, u \preceq \tau \vdash *}\ (\preceq\ \mathit{var}) \qquad \frac{\Gamma \vdash A \quad \Gamma, \Gamma' \vdash *}{\Gamma, \Gamma' \vdash A}\ (\mathit{weak})$$

where $\Gamma \vdash A$ is any judgment.

Table 2. Rules for Contexts.

3.2 Subtyping

The subtyping rules are listed in Table 3. The rules for constants, reflexivity, transitivity and for the arrow-type constructor (that behaves contravariantly in its domain with respect to the $\preceq$ relation) are standard: the two rules related to *width* subtyping over class-types are discussed below.

The $(\preceq_{shift})$ rule allows methods (together with their types) to be moved from the interface–row to the completion–row of a class–type. The $(\preceq_{hide})$ rule is the classical rule of subtyping in "width" that allows generalizing a class–type to other class–types containing fewer methods (types). The condition $\overline{p} \notin \mathcal{L}(\overline{\alpha})$[4] ensures that the remaining methods do not use the methods $\overline{p}$.

$$\frac{\Gamma \vdash * \quad u \preceq \tau \in \Gamma}{\Gamma \vdash u \preceq \tau}\ (\preceq \mathit{proj}) \qquad \frac{\Gamma \vdash *}{\Gamma \vdash \tau \preceq \tau}\ (\preceq \mathit{refl})$$

$$\frac{\Gamma \vdash \sigma \preceq \tau \quad \Gamma \vdash \tau \preceq \rho}{\Gamma \vdash \sigma \preceq \rho}\ (\preceq \mathit{trans}) \quad \frac{\Gamma \vdash \sigma' \preceq \sigma \quad \Gamma \vdash \tau \preceq \tau'}{\Gamma \vdash \sigma \to \tau \preceq \sigma' \to \tau'}\ (\preceq \mathit{arrow})$$

$$\frac{\Gamma \vdash *}{\Gamma \vdash \mathtt{class}\, t.\langle\!| R \mid m{:}\alpha |\!\rangle \circ C \preceq \mathtt{class}\, t.R \circ \langle\!| C \mid m{:}\alpha |\!\rangle}\ (\preceq_{shift})$$

$$\frac{\Gamma \vdash * \quad \overline{p} \notin \mathcal{L}(\overline{\alpha})}{\Gamma \vdash \mathtt{class}\, t.\langle\!| \overline{m{:}\alpha} |\!\rangle \circ \langle\!| C \mid \overline{p{:}\gamma} |\!\rangle \preceq \mathtt{class}\, t.\langle\!| \overline{m{:}\alpha} |\!\rangle \circ C}\ (\preceq_{hide})$$

where $\mathcal{L}(\overline{\alpha})$ denotes the set of method names occurring in the labels of $\overline{\alpha}$, i.e: $\mathcal{L}(\tau_\Delta)=\{\Delta\}$, and $\mathcal{L}(\overline{\alpha}, \tau_\Delta)=\mathcal{L}(\overline{\alpha}) \cup \Delta$.

Table 3. Subtyping Rules.

[4] The vector notation $\bar{\ }$ has the usual meaning.

Notice that, although ($\preceq_{hide}$) hides only methods which are in the completion–row, the combination of ($\preceq_{shift}$) and ($\preceq_{hide}$) allows methods in the interface–row to be hidden, by first moving them to the completion-row. Hence, the combination of ($\preceq_{shift}$) and ($\preceq_{hide}$) leads to the same subtype relation over class-types as in [7]: a set of methods may be *hidden* from the interface–row of a class–types only if no method in the set occurs in the dependency set of the remaining methods of the interface. Since labels are enforced to provide a sound representation of the dependencies of a method (see the (ext_{ext}) rule in the next subsection), hiding of methods may safely be done looking at the method labels without imposing the covariance constraints on the occurrences of the bound variable peculiar to the standard subtype rules for recursive record-types.

3.3 Typing Rules

The full set of typing rules is presented in Table 4. The rules (*proj*), (*exp appl*) and (*exp abs*) are standard; the subsumption rule ($\preceq$) is used in conjunction with the subtype relation to account to type promotion. The remaining rules are described next.

The rule ($\langle\rangle$) should be self-explanatory: since the empty object contains no method, it needs no further method to be completed.

$$\frac{\Gamma \vdash * \quad x:\tau \in \Gamma}{\Gamma \vdash x:\tau}\ (proj) \qquad \frac{\Gamma,\, x{:}\tau_1 \vdash e{:}\tau_2}{\Gamma \vdash \lambda x.e{:}\tau_1 \to \tau_2}\ (exp\ abs)$$

$$\frac{\Gamma \vdash e_1{:}\tau_1 \to \tau_2 \quad \Gamma \vdash e_2{:}\tau_1}{\Gamma \vdash e_1 e_2{:}\tau_2}\ (exp\ appl) \qquad \frac{\Gamma \vdash e:\sigma \quad \Gamma \vdash \sigma \preceq \tau}{\Gamma \vdash e:\tau}\ (\preceq)$$

$$\frac{\Gamma \vdash *}{\Gamma \vdash \langle\rangle : \mathtt{class}\,t.\langle\!\rangle \circ \langle\!\rangle}\ (\langle\rangle) \qquad \frac{\Gamma \vdash e:\sigma \quad \Gamma \vdash \sigma \preceq \mathtt{class}\,t.\langle \overline{m{:}\alpha}, n{:}\tau_{\{\overline{m}\}} \rangle \circ \langle\!\rangle}{\Gamma \vdash e \Leftarrow n : [\sigma/t]\tau}\ (send)$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \mathtt{class}\,t.R \circ C \quad \overline{m{:}\alpha} \in R \circ C \quad n, \overline{p} \notin \mathcal{M}(R) \cup \mathcal{M}(C) \\ \Gamma, u \preceq \mathtt{class}\,t.\langle \overline{m{:}\alpha}, \overline{p{:}\gamma}, n{:}\tau_{\{\overline{m},\overline{p}\}} \rangle \circ \langle\!\rangle \vdash e_2 : [u/t](t \to \tau)\end{array}}{\Gamma \vdash \langle e_1 \leftarrow\!\circ\ n{=}e_2 \rangle : \mathtt{class}\,t.\langle R \mid n{:}\tau_{\{\overline{m},\overline{p}\}} \rangle \circ \langle C \mid \overline{p{:}\gamma} \rangle}\ (ext_{ext})$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \sigma \quad \Gamma \vdash \sigma \preceq \mathtt{class}\,t.\langle\!\rangle \circ \langle \overline{m{:}\alpha}, n{:}\tau_{\{\overline{m}\}} \rangle \\ \Gamma, u \preceq \mathtt{class}\,t.\langle \overline{m{:}\alpha}, n{:}\tau_{\{\overline{m}\}} \rangle \circ \langle\!\rangle \vdash e_2 : [u/t](t \to \tau)\end{array}}{\Gamma \vdash \langle e_1 \leftarrow\!\circ\ n{=}e_2 \rangle : \sigma}\ (ext_{over})$$

$$\frac{\Gamma \vdash e : \mathtt{class}\,t.\langle n{:}\tau_{\{\overline{m}\}} \rangle \circ \langle \overline{m{:}\alpha} \rangle \quad \Gamma \vdash \sigma \preceq \mathtt{class}\,t.\langle \overline{m{:}\alpha}, n{:}\tau_{\{\overline{m}\}} \rangle \circ \langle\!\rangle}{\Gamma \vdash e \hookleftarrow n : [\sigma/t](t \to \tau)}\ (search)$$

Table 4. Typing Rules.

The intuitive reading of the (*send*) rule is as follows: according to the subtype relation, the type σ above will, in general, have the form[5] $\mathtt{class}\,t.\langle R \mid$

[5] There is a subtler point here, that explains the use of the generic type σ instead

$\overline{m{:}\alpha}, n{:}\tau_{\{\overline{m}\}}\rangle\!\rangle \circ C$, for any R and C, provided that no method-name of R and C occurs in the labels of either $\overline{m}$ or n. Accordingly, in order to type a method invocation for an n method on an object e, we require (i) that e contains (in its interface–row) the method-name n, and (ii) that every method contained in the label associated to the type of n is also contained in the interface-row of the type of e. The substitution for t in τ in the conclusion of the rule reflects, as in [14], the recursive nature of class-types.

To explain the typing rule for method addition, we distinguish the case when the n method to be added does not occur in type of the object that gets extended, from the case when it does.

In the first case, we need to determine the labeled-type of n, and possibly to extend the completion-row with new methods referenced to by this method. This is accomplished by the rule (ext_{ext}), where $\overline{m{:}\alpha} \in R \circ C$ indicates that the $\overline{m{:}\alpha}$ methods are contained in $R \circ C$, whereas the condition $n, \overline{p} \notin \mathcal{M}(R) \cup \mathcal{M}(C)$ ensures that the final type will be well-formed. The intuitive reading of the rule is as follows. First we note that n may, in general, depend on methods that are already contained in the object as well as on methods that are yet to be added. Accordingly, the label associated to the type of n includes the $\overline{m}$ methods that are already present in the type of e_1, and the $\overline{p}$ that are, instead, new. Note, further, that all of these methods (i.e. the $\overline{m}$ and $\overline{p}$ methods) are assumed to occur in the interface-row of the type that constrains u in the typing of e_2: this guarantees that the choice of $\{\overline{m}, \overline{p}\}$ as the label of n is a sound representation of the dependencies of n. To see this, consider the case when $e_2 = \lambda self.(self \Leftarrow p)$, for a given method p. Then, an inspection of the ($send$) rule shows that, in order for the invocation $self \Leftarrow p$ to be typeable, the interface-row of the type of $self$ must include not only p, but also all of the, say, $\overline{q}$ methods in the label of the type of p. But then, the label of n must include p, a direct reference, as well as the $\overline{q}$ methods that n references indirectly via p. Note, finally, that, as in [14], the type of n has the form $t \rightarrow \tau$ (with a class-type substituted for t) to conform with the self-application semantics of method invocation. The difference is in the way polymorphic types of method bodies are instantiated to allow applications to extended objects. Instead of introducing row-variables, we allow applications of e_2 to any object of type u with u subtype of $\texttt{class}\, t.\langle\!\langle \overline{m{:}\alpha}, \overline{p{:}\gamma}, n{:}\tau_{\{\overline{m},\overline{p}\}} \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$.

The other case of method addition arises when the n method occurs in the type of the object e_1 that is being extended, and it is handled by the rule (ext_{over}). There are two possible situations that may lead to this case: either n has already been added to e_1 (in which case, the addition is, operationally, an override) or it is referenced to by other methods of that object. In the first case, n occurs in the interface–row of the type of e_1, in the second in the completion–row. However, as we anticipated, these two situations may be dealt with uniformly by assuming that n occurs in the completion-row, where it can be moved by an application of ($\preceq_{shift}$). Similarly to the case of the ($send$) rule, e_1 above

of the indicated type. The point is that when e is a variable (e.g. $self$) σ may as well be a constrained type-variable occurring in the context Γ. This allows method invocations inside the bodies of methods.

will, in general, have the type $\vdash e_1 : \mathtt{class}\, t.R\circ C$, with $\overline{m{:}\alpha}, n{:}\tau_{\{\overline{m}\}} \in R\circ C$, where the n and $\overline{m}$ methods do not depend on other methods of $R\circ C$ (this is ensured by the choice of $\overline{m}$ as the label of n, made when typing e_1). The constraint for σ is then motivated by the fact that every type that satisfies these constraints is a subtype of $\mathtt{class}\, t.\langle\!\rangle\circ\langle\!\overline{m{:}\alpha}, n{:}\tau_{\{\overline{m}\}}\!\rangle$. Finally, as for (*send*), the generality that derives from the use of the type σ is needed to carry out derivations in which the (ext_{over}) rule is applied when e_1 is a variable (e.g. *self*). Propagation of labels may be observed as in the example above, now taking $e_2 = \lambda self.\langle self \leftarrow\!\circ\; m{=}\lambda s.(s \Leftarrow p)\rangle$ where p is, say, a constant method (whose type has an empty label).

We conclude with the rule (*search*) for typing a search expression. The intuitive reading of the rule is as follows: first note that the $e \hookleftarrow n$ expression is error-free only if e is an object that contains the n method and its dependencies; when this is the case, the result of $e \hookleftarrow n$ is the body defined by the last addition of the n method. Then, it follows that in order for $e \hookleftarrow n$ to be typeable, the type of e must contain n as well as its dependencies $\overline{m{:}\alpha}$, with the additional constraint that n must occur in the interface-row of the type, so as to guarantee that e does indeed contain the n method. This explains the left judgment in the premises of the above rule; as for the remaining judgment, since the result of evaluating $e \hookleftarrow n$ is the body defined in the last addition of n to e, its type may be chosen to be any instance of the type we deduced for this body when it was added (see the (ext_{ext}) rule).

4 Examples

The following two examples help illustrate the distinguishing features of our type system, and relate it to previous proposals (see [5] for other examples). To ease the presentation, we use $\langle x = e\rangle$ as short for $\langle\langle\rangle \leftarrow\!\circ\; x = e\rangle$ and we assume that omitted labels represent the empty set.

Contexts

$$\Gamma_0 = u \preceq \mathtt{class}\, t.\langle\!\mathtt{x}{:}int, \mathtt{mv}{:}(int \to t)_{\{\mathtt{x}\}}\!\rangle\circ\langle\!\rangle \qquad \Gamma_1 = \Gamma_0, \mathtt{self} : u, \mathtt{dx} : int$$
$$\Gamma_2 = \Gamma_1, v \preceq \mathtt{class}\, t.\langle\!\mathtt{x}{:}int\!\rangle\circ\langle\!\rangle \qquad \Gamma_3 = \Gamma_2, \mathtt{s} : v$$

Derivation

1. $\Gamma_3 \vdash (\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx} : int$
   by (*send*) from $\Gamma_3 \vdash \mathtt{self} : u$, and $\Gamma_3 \vdash u \preceq \mathtt{class}\, t.\langle\!\mathtt{x}{:}int\!\rangle\circ\langle\!\rangle$.
2. $\Gamma_2 \vdash \lambda\mathtt{s}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx} : v \to int$
3. $\Gamma_1 \vdash \langle \mathtt{self} \leftarrow\!\circ\; \mathtt{x} = \lambda\mathtt{s}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx}\rangle : u$
   by (ext_{over}) from $\Gamma_1 \vdash \mathtt{self} : u$, $\Gamma_1 \vdash u \preceq \mathtt{class}\, t.\langle\!\rangle\circ\langle\!\mathtt{x}{:}int\!\rangle$, and 2.
4. $\Gamma_0 \vdash \lambda\mathtt{self}.\lambda\mathtt{dx}.\langle \mathtt{self} \leftarrow\!\circ\; \mathtt{x} = \lambda\mathtt{s}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx}\rangle : u \to int \to u$
5. $\varepsilon \vdash \mathtt{ip} : \mathtt{class}\, t.\langle\!\mathtt{mv}{:}(int \to t)_{\{\mathtt{x}\}}\!\rangle\circ\langle\!\mathtt{x}{:}int\!\rangle$
   by (ext_{ext}) from $\varepsilon \vdash \langle\ \rangle : \mathtt{class}\, t.\langle\!\rangle\circ\langle\!\rangle$, and 4.

Table 5.

Example 1. This example shows that our typing rules allow complete freedom in the order of method additions. Let ip be the following incomplete object:

$$\texttt{ip} = \langle \texttt{mv} = \lambda\texttt{self}.\lambda\texttt{dx}.\langle \texttt{self} \leftarrow\!\!\circ\, \texttt{x} = \lambda\texttt{s}.(\texttt{self} \Leftarrow \texttt{x}) + \texttt{dx}\rangle\rangle.$$

While this object *cannot* be typed in the system of [14], Table 5 shows that ip is typeable in our system. From this, we may easily derive the expected judgment:

$$\varepsilon \vdash \langle \texttt{ip} \leftarrow\!\!\circ\, \texttt{x} = \lambda\texttt{self}.3\rangle : \texttt{class}\, t.\langle\!\langle \texttt{move}:(int \to t)_{\{\texttt{x}\}}, \texttt{x}:int \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle.$$

Example 2. This example illustrates one interesting difference between our system and a related extension of the system of [14] presented in [15]. In this latter paper, subtyping arises from introducing two distinguished sets of object-types: pro-types, and obj-types. These types are ordered by the subtype relation, so as to allow pro-typed objects to be "packaged" to produce corresponding obj-typed objects.

Objects having pro-types may be freely operated with (they may be sent messages, or extended with new methods, or modified by overriding existing methods), but only trivial subtyping is allowed over pro-types. On the other hand, objects having obj-types may only respond to messages, or modify their own structure from the "inside" (i.e. via overrides on *self* within their own methods), whereas they may *not* be modified or extended from the outside.

Preventing from outside extension and override allows "width" and "depth" subtyping for obj-types, provided that the bound type-variable of an obj-type does not occur in contravariant position. This distinction between pro- and obj-types has other interesting consequences: first it gives insights into the different nature of the inheritance and client interfaces of objects and classes in object-oriented languages; secondly, as shown in [15], it allows a quite natural modeling of method encapsulation. However, the resulting type discipline does not allow to type some expressions that we can deal with. To illustrate the problem, consider the following function:

$$\texttt{plot} \overset{def}{=} \lambda\texttt{p}.\langle \texttt{p} \leftarrow\!\!\circ\, \texttt{c}{=}\lambda\texttt{s}.\texttt{white}\rangle,$$

which can be viewed as a mapping of one-dimensional points to colored-points. The following judgment is easily derived in our type system:

$$\texttt{plot} : \texttt{class}\, t.\langle\!\langle \texttt{x}:int \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle \to \texttt{class}\, t.\langle\!\langle \texttt{x}:int, \texttt{c}:col \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle.$$

Then, given a colored point cp of type, say, $\texttt{class}\, t.\langle\!\langle \texttt{x}:int, \texttt{c}:col \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$, we may safely apply plot to cp because, by subtyping, we have $\texttt{cp} : \texttt{class}\, t.\langle\!\langle \texttt{x}:int \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$.

This simple property is lost in the system of [15]. In fact, having distinguished obj- and pro-types, we may prove that:

$$\texttt{plot} : \texttt{pro}\, t.\langle\!\langle \texttt{x}:int \rangle\!\rangle \to \texttt{probj}\, t.\langle\!\langle \texttt{x} : int, \texttt{c}:col \rangle\!\rangle,$$

where probj is either pro or obj, but we *cannot* prove that:

$$\texttt{plot} : \texttt{obj}\, t.\langle\!\langle \texttt{x}:int \rangle\!\rangle \to \texttt{probj}\, t.\langle\!\langle \texttt{x} : int, \texttt{c}:col \rangle\!\rangle.$$

This is because plot modifies its input argument with a method addition, an operation that is only allowed on pro-types. But, then, there is no way that we may type an application of plot to the colored point cp. In fact, we may either take cp to have type $\texttt{obj}\, t.\langle\!\langle \texttt{x}:int \rangle\!\rangle$ or type $\texttt{pro}\, t.\langle\!\langle \texttt{x}:int, \texttt{c}:col \rangle\!\rangle$, but, according to the subtype relation of [15], neither of these types is a subtype of $\texttt{pro}\, t.\langle\!\langle \texttt{x}:int \rangle\!\rangle$ (since pro-types are subtypes of obj-types, but not vice-versa, and "width" subtyping is not allowed over pro-types).

5 Soundness of the Type System

We conclude the description of the type system stating soundness. We first show that types are preserved by the reduction process. Due to the lack of space, we can only state the result: the reader is referred to [5] for a detailed proof of the following theorem.

Theorem 1 (Subject Reduction). *If $\Gamma \vdash e_1 : \tau$ is derivable and $e_1 \overset{eval}{\twoheadrightarrow} e_2$, then $\Gamma \vdash e_2 : \tau$ is also derivable.* □

The subject reduction property shows the power of the type system. Labeled-types not only allow a restricted form of subtyping that enriches the set of typeable objects, but they also fit well with the operational semantics based on the $\hookleftarrow$ operation: in fact, the typing rule for the $\hookleftarrow$ expression is based on the information given by the labels.

Since an $\overset{eval}{\longrightarrow}$ step produces the object *err* (which has no type) when a message m is sent to an expression which does not define an object with a method m, the type soundness follows directly from Theorem 1.

Theorem 2 (Type Soundness). *If $\varepsilon \vdash e : \tau$ is derivable for some τ, then the evaluation of e cannot produce err, i.e. $e \overset{eval}{\not\twoheadrightarrow} err$.* □

6 Conclusion

We have presented an extension of the *Lambda Calculus of Objects* [14] with a new type system that gives provision both for incomplete objects, in the style of [6], and for a relation of "width" subtyping in the style of [7].

The main technical tool of the new system is represented by labeled-types, that are central both to the subtype relation and to the rendering of method polymorphism based on bounded quantification. While it could be argued that labels may be costly to handle and somehow difficult to explain, it should be noticed that their use is relevant to the soundness of the system and not to the meaning of types. Notice, in fact, that labels are *introduced* (i.e. computed) upon object extension, in the (ext_{ext}) rule, and then only *used* in the rules (ext_{over}), $(send)$, $(search)$ and in the rules that define the subtype relation. In particular, labels are computed in the application of (ext_{ext}) by looking at the set of method-names occurring in the bound of the variable u used in the typing of method bodies. Therefore, in principle, the system can infer labels automatically, and then simply verify that the application of the rules (ext_{over}), $(send)$, and $(\preceq)$ do respect them. In this way labels would be made transparent from "outside" class-types (hence, to the user of the system) and would only serve as "internal" devices needed to ensure type soundness.

A system that exhibits features comparable to our system is Baby Modula-3 [1] which, however, we generalize in two respects: (*a*) we allows object extensions and subsumptions in any order, while Baby Modula-3 requires that all the extensions be done before the subsumptions; (*b*) our completions may be extended as a result of a method addition, while in Baby Modula-3 completions are fixed ahead of time, prior to any addition. A feature of [1] which we do not provide, even though we could, is the distinction between fields and methods, that allows one to isolate the state of an object from the operations on the state.

Another related paper is [20], which combines row-variables and refined subtyping in presence of extensible objects. There are similarities with our proposal, in particular in that our interface and completion-rows behave similarly to the Pre types and Maybe types of [20]. On the other side, the subtyping of [20] is weaker than ours, since for example one cannot derive that the type "colored point" a subtype of "point", i.e. that $\texttt{class}\,t.\langle\!\langle \mathrm{x}{:}int, \mathrm{c}{:}col \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle \preceq \texttt{class}\,t.\langle\!\langle \mathrm{x}{:}int \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle$ using our notation. Also, unlike [20], we do not require object-types to be total functions from names to types and we disregard row-variables by taking advantage of subtyping. Finally, our system appears to be more liberal in the typing of objects, since we allow incomplete objects to be typed, and the same method to be hidden and later safely redefined with a possibly incompatible type, two features that are not accounted for in [20].

Further remarks concern [15]. On one side, we allow to do extensions, overrides and subtypings on the same object in any order, while [15] forbids to extend or override objects for which we already used subtyping. On the other, method encapsulation is not accounted in our system and is instead provided in [15]. To this regard, we note that the solution proposed in [15] could be accommodated just as well in our system. As in [15], we would need to distinguish the types of prototypes from the types of objects, so as to allow altering the structure of the former with method additions and overrides while instead preventing such operations to be applied to the latter. Methods of a complete prototype (i.e. a prototype whose completion-row is empty) could then be "sealed" (hence encapsulated) within the object corresponding to the prototype exactly as in the system of [15].

A few other studies on delegation-based languages have recently been proposed as elaborations of the Lambda Calculus of Objects and related calculi:

[19] presents a (decidable) typed version of the original calculus of [14];
[18] adds object extension and *width* subtyping to the system of [3];
[4] presents a type system for the Lambda Calculus of Objects based on *matching*.

In particular, the system of [4] and the one of this paper share the same idea of using bounded type-variables to capture polymorphic method-types. The key difference is that [4] uses a simplified notion of *matching* [8, 2] (without subsumption) and *match*-bound variables, whereas here we use subtyping and *subtype*-bound variables.

It should also be mentioned that [4] proves that bounded type-variables and row-variables have the same expressive power, more precisely the systems of [4] and [14] derive the same judgments from the empty basis[6]. Instead there is a trade-off between the present system and that of [14], since on one side we add

[6] There are also examples suggesting that bounded type-variables can replace row-variables when the context is not empty, like the following judgment typing composition of two messages:

$$v \preceq \texttt{class}\,t.\langle\!\langle m : \tau \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle, u \preceq \texttt{class}\,t.\langle\!\langle n : v \rangle\!\rangle \circ \langle\!\langle\rangle\!\rangle \vdash \lambda x.(x \Leftarrow n) \Leftarrow m : u \to \tau.$$

subtyping, while on the other side, the use of labeled types prevents us from deriving some judgments which are valid in [14].

Acknowledgments. The present version of this paper has strongly benefitted from comments and remarks by an anonymous referee that the authors like to thank.

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. On Subtyping and Matching. In *ECOOP'95*, *LNCS* 952, 145–167. Springer–Verlag, 1995.
3. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer–Verlag, 1996.
4. V. Bono and M. Bugliesi. Matching Constraints for the Lambda Calculus of Objects. In *TLCA'97*, *LNCS*. Springer-Verlag, 1997. To appear.
5. V. Bono, M. Bugliesi, M. Dezani-Ciancaglini, and L. Liquori. Subtyping Constraints for Incomplete Objects. Technical Report CS-34-97, Computer Science Department, Turin University, 1996.
6. V. Bono, M. Bugliesi, and L. Liquori. A Lambda Calculus of Incomplete Objects. In *MFCS'96*, *LNCS* 1113, 218–229. Springer-Verlag, 1996.
7. V. Bono and L. Liquori. A Subtyping for the Fisher-Honsell-Mitchell Lambda Calculus of Objects. In *CSL'94*, *LNCS* 933, 16–30. Springer-Verlag, 1995.
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. L. Cardelli. A Semantics of Multiple Inheritance. *Information and Computation*, 76:138–164, 1988.
10. L. Cardelli and J.C. Mitchell. Operations on Records. *Mathematical Structures in Computer Sciences*, 1(1):3–48, 1991.
11. L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction and Polymorphism. *Computing Surveys*, 17(4):471–522, 1985.
12. W. Cook, W. Hill, and P. Canning. Inheritance is not Subtyping. In *POPL'90*, 125–135. ACM Press, 1990.
13. W.R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.
14. 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.
15. K. Fisher and J. C. Mitchell. A Delegation-based Object Calculus with Subtyping. In *FCT'95*, *LNCS* 965, 42–61. Springer-Verlag, 1995.
16. A. Goldberg and D. Robson. *Smalltalk-80, The Language and its Implementation.* Addison Wesley, 1983.
17. R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *J.ACM*, 40(1):143–184, 1993.
18. L. Liquori. An Extended Theory of Primitive Objects. Technical Report CS-23-96, Computer Science Department, Turin University, 1996.
19. L. Liquori and B. Castagna. A Typed Lambda Calculus of Objects. In *Asian'96*, *LNCS* 1179, 129–141. Springer-Verlag, 1996.
20. D. Rémy. Refined Subtyping and Row Variables for Record Types. Draft, 1995.
21. D. Ungar and R. B. Smith. Self: the Power of Simplicity. In *OOPSLA'87*, 227–241. ACM Press, 1987.
22. M. Wand. Complete Type Inference for Simple Objects. In *LICS'87*, 37–44. Silver Spring, 1987.