A Lambda Calculus of Incomplete Objects

Viviana Bono* **Michele Bugliesi**** **Luigi Liquori***

*Dipartimento di Informatica, Università di Torino
C.so Svizzera 185, I-10149 Torino, Italy
e-mail: {bono,liquori}@di.unito.it

**Dipartimento di Matematica, Università di Padova
Via Belzoni 7, I-35131 Padova, Italy
e-mail: michele@math.unipd.it

Abstract. This paper extends the Lambda Calculus of Objects as proposed in [5] with a new support for *incomplete* objects. Incomplete objects behave operationally as "standard" objects; their typing, instead, is different, as they may be typed even though they contain references to methods that are yet to be added. As a byproduct, incomplete objects may be typed independently of the order of their methods and, consequently, the operational semantics of the untyped calculus may be soundly defined relying on a permutation rule that treats objects as sets of methods. The new type system is a conservative extension of the system of [5] that retains the *mytype* specialization property for inherited methods peculiar to [5], as well as the ability to statically detect run-time errors such as *message not understood*.

1 Introduction

Object-oriented languages have been classified as either *class-based* or *delegation-based* according to the underlying object-oriented model. In class-based languages, such as *Smalltalk* [7] and *C++* [4], the implementation of an object is specified by a template, the class of the object, and every object is created by instantiating its class. In contrast, delegation-based languages, such as *Self* [9], are centered around the idea that objects are created dynamically by modifying existing objects used as *prototypes*. An object created from a given prototype may add new methods or redefine methods supplied by the prototype, and any message sent to the object is handled directly by that object, if it contains a corresponding method, or it is "passed back", i.e. *delegated*, to the prototype.

Delegation-based languages have gained renewed popularity in the last years. In [1], an object calculus is presented that supports (destructive) method override and inheritance by *object subsumption*. In [5], a functional model of a delegation-based calculus is presented that extends previous foundational work from [8]. The *Lambda Calculus of Objects* of [5] is an untyped lambda 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. The resulting calculus allows a natural encoding of the object-oriented notion of *self* directly by lambda abstraction, and it provides a powerful and simple inheritance mechanism based on a dynamic method-lookup semantics. Furthermore, the static type system supports an elegant form of *mytype* method specialization whereby the type of inherited methods may be specialized to the type of the inheriting objects. Subsequent work [3,6] on this calculus has shown that the type system is amenable to extensions with different forms of subtyping.

In the present paper we extend the original calculus of [5] along a direction orthogonal to subtyping. One weakness of [5] is that the typing rules impose a rather rigid discipline on the way that objects may be created from a prototype. In particular, the addition of an m method to an object can be typed correctly only if all the methods that are referenced to (via message sends or method overrides to *self*) in the body of m are already available from that object. Besides making it difficult to write mutually recursive methods, this constraint leads to a somewhat involved formulation of the operational semantics where a notion of *objects in standard form* is needed to extract the appropriate method upon evaluation of a message.

Our extension is based on a new encoding of the types of objects that allows us to treat objects as *sets of methods* (as opposed to ordered sequences) and, consequently, to rely on a simpler operational semantics. The new encoding also gives additional flexibility to the type system, by allowing a method invocation to be typed correctly even though the receiver of the message is an *incomplete* object, i.e. an object that contains references to methods that are yet to be added. This flexibility appears to be desirable for prototyping languages, such as delegation-based languages, where prototypes may reasonably be defined, and operated with as well, while part of their implementation (i.e. their methods) are yet to be defined. The new features are achieved at the expense of little additional complexity in the typing rules, and they provide a conservative extension of the original system: the new type system retains the property of method specialization of [5], as well as the ability to statically detect run-time errors such as *message not understood.*

The rest of the paper is organized as follows. In Section 2 we briefly overview the untyped calculus of [5] and present the new operational semantics. In Section 3 we present the new typing rules for objects. In Section 4 we first prove Subject Reduction and then use it to show Type Soundness. We conclude in Section 5 with some final remarks.

2 The Untyped Calculus

The syntax of the untyped calculus is as in [5]. An expression can be any of the following:

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

where x is a variable, c a constant and m a method name. The object-related forms are as in [5], namely:

$\langle\rangle$	is the empty object,
$e \Leftarrow m$	sends message m to object e,
$\langle e_1 \leftarrow\!\!+\, m{=}e_2\rangle$	extends object e_1 with a new method m having body e_2,
$\langle e_1 \leftarrow m{=}e_2\rangle$	replaces e_1's method body for m with e_2.

The expression $\langle e_1 \leftarrow\!\!+\, m{=}e_2\rangle$ is defined only when e_1 denotes an object that does not have an m method, whereas $\langle e_1 \leftarrow m{=}e_2\rangle$ is defined only when e_1 denotes an object that *does* contain an m method. Both these conditions are enforced statically by the type system.

The other main operation on object is method invocation. Methods are invoked by means of message sends according to the following semantics: when the object $\langle e_1 \leftarrow\!\!+\, m{=}e_2\rangle$ is sent the message m, the result is obtained by "self-applying" e_2 to $\langle e_1 \leftarrow\!\!+\, m{=}e_2\rangle$. Therefore, in defining the operational semantics of the calculus, we must give, besides the rules of β-reduction and method invocation, also a mechanism for extracting the appropriate method of an object. As it turns out, the following three rules suffice ($\leftarrow\!\!\circ$ denotes either $\leftarrow\!\!+$ or $\leftarrow$):

$$\begin{array}{lll}
(\beta) & (\lambda x.e_1)\, e_2 & \stackrel{ev}{\to} [e_2/x]\, e_1 \\
(\Leftarrow) & \langle e_1 \leftarrow\!\!\circ\, m{=}e_2\rangle \Leftarrow m & \stackrel{ev}{\to} e_2\, \langle e_1 \leftarrow\!\!\circ\, m{=}e_2\rangle \\
(\mathit{perm}) & \langle\langle e \leftarrow\!\!\circ\, m{=}e_1\rangle \leftarrow\!\!\circ\, n{=}e_2\rangle & \stackrel{ev}{\to} \langle\langle e \leftarrow\!\!\circ\, n{=}e_2\rangle \leftarrow\!\!\circ\, m{=}e_1\rangle.
\end{array}$$

Note that we allow permutations only for methods with different names, whereas different definitions for the same name maintain their respective position through subsequent overrides. Accordingly, a message send for any m method always selects the definition provided by the last override for that method.

The operational semantics of the calculus is defined as the least reflexive, transitive and contextual closure $\stackrel{ev}{\twoheadrightarrow}$ generated by the reduction rules above. As a remark, we note that the (*perm*) rule above is justified in our calculus since the following equational rule for objects is sound with respect to the type system:

$$\langle\langle e \leftarrow\!\!\circ\, m{=}e_1\rangle \leftarrow\!\!\circ\, n{=}e_2\rangle = \langle\langle e \leftarrow\!\!\circ\, n{=}e_2\rangle \leftarrow\!\!\circ\, m{=}e_1\rangle.$$

This equality did not hold for the system of [5] because in that case the typing rules allow objects to be typed only when methods are added in the appropriate order. As a consequence, in order for method extraction to be performed correctly, a series of *bookkeping* rules are needed, that make the definition of the operational semantics rather involved.

3 Static Type System

The type of an incomplete object is defined by a type expression of the form:

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

where the m_i's and p_i's are method names, and the α_i's and the γ_i's are *labeled-type* expressions (whose role is discussed below). Given the above 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 order of methods within each row is irrelevant and we rely on the following equational rule for rows throughout the paper:

$$\langle\!\langle \langle\!\langle R \mid n{:}\tau_1 \rangle\!\rangle \mid m{:}\tau_2 \rangle\!\rangle = \langle\!\langle \langle\!\langle R \mid m{:}\tau_2 \rangle\!\rangle \mid n{:}\tau_1 \rangle\!\rangle.$$

The binder `class` scopes over the two rows of the type, and the bound variable t may occur free within the scope of the binder, with every free occurrence referring to the class-type itself; thus, as in [5], class-types are a form of recursively-defined types.

The intuitive reading of these types is as follows. The interface-row describes all the methods (and their types) that are contained in the objects of the current type, and that may be invoked by means of corresponding messages. The completion-row, instead, lists the methods (and their types) that are referenced to by the methods of the object (whose types are listed in the interface-row) even though they are not yet available from the object. Accordingly, given an object containing the m_i methods of the interface-row, the completion-row lists the methods (and associated types) that are needed to "complete" the object. As we anticipated, this encoding of types leads us to formulate typing rules that allow object expressions to be formed by sequences of method additions and overrides where the order of such operations does not matter. Using labeled-types within the rows of our class-types, also enables us to type a method invocation even though the receiver of the message is incomplete (i.e., its class-type has a non-empty completion-row).

Labeled-types were first introduced in [3], to model a form of *width* subtyping for the original calculus of [5]. Here they bear (almost) the same meaning: if τ_Δ is the type of, say, an m method within an object, then Δ provides an (approximate) representation of the remaining methods of that object upon which m depends: in other words, Δ includes 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.

The use of transitive (or indirect) references within labels is enforced by the typing rules and, as we shall se in Section 3.3, it is crucial to ensure the correctness of method invocation on incomplete objects. Having direct as well as indirect references within labels, the typing rule for a method invocation may be stated as follows:

$$\frac{\Gamma \vdash e : \texttt{class}\, t.\langle\!\langle R \mid \mathbf{n}{:}\boldsymbol{\alpha}, m{:}\tau_{\{\mathbf{n}\}} \rangle\!\rangle \bullet C}{\Gamma \vdash e \Leftarrow m : [(\texttt{class}\, t.\langle\!\langle R \mid \mathbf{n}{:}\boldsymbol{\alpha}, m{:}\tau_{\{\mathbf{n}\}} \rangle\!\rangle \bullet C)/t]\tau} \quad (\mathit{send})$$

In order to type a method invocation for an m method in an object e, we require (i) that e contain (in its interface-row) the method name m, and (ii) that all of the methods contained in the label associated with m be contained in the interface-row of the object's type.

3.1 Types, Rows, and Kinds

The type expressions include type-constants, type variables, function-types and class-types. The symbols α, β, $\ldots$ range over labeled-types. We also use the notation $\mathbf{m{:}\alpha}$, as shorthand for $m_1{:}\alpha_1, \ldots, m_k{:}\alpha_k$, for some k. The sets of types, rows and kinds are defined by the following productions:

Types	$\tau ::= \iota \mid t \mid \tau \to \tau \mid \mathtt{class}\, t.R \bullet R$
Rows	$R ::= r \mid \langle\!\langle\rangle\!\rangle \mid \langle\!\langle R \mid m{:}\tau_\Delta \rangle\!\rangle \mid R\tau \mid \lambda t.R \qquad (m \notin \Delta)$
Labels	$\Delta ::= \{m_1, \ldots, m_k\} \qquad (k \geq 0)$
Kinds	$\kappa ::= T \mid [m_1, \ldots, \mathtt{m}_k] \mid T^n \to [m_1, \ldots, m_k] \quad (n \geq 1,\ k \geq 0).$

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 notations, namely: R and r to denote respectively interface-rows and interface-row variables, whereas C and c to stand for arbitrary completion-rows and completion-row variables.

Our definition of class-type generalizes the original definition of [5] in that types of the form $\mathtt{class}\, t.R$ from [5] are represented here simply as $\mathtt{class}\, t.R \bullet \langle\!\langle\rangle\!\rangle$. As we shall see, a corresponding generalization applies to the typing of method bodies: a method body will be built as a polymorphic function whose type is defined in terms of a (universally quantified) row variable, as in [5], and of a (universally quantified) completion variable.

This generalization requires a few changes in the type system, in order to characterize the interdependence between the interface- and completion-rows of a class-type: the intention is to give a more precise definition of the type $\mathtt{class}\, t.R \bullet C$ by requiring that R and C be *disjoint*, i.e. that methods occurring in R do not occur in C and vice-versa. To formalize this idea, we redefine the meaning of the kinds $[m_1, \ldots, m_k]$ and $T^n \to [m_1, \ldots, m_k]$ as follows. The elements of the kind $[m_1, \ldots, m_k]$ are pairs of disjoint rows neither of which contains any of the method names $m_1, \ldots, m_k$. A corresponding interpretation applies to the kinds $T^n \to [m_1, \ldots, m_k]$, for $n \geq 1$, that are used to infer polymorphic types for method bodies.

The structure of valid context (see Appendix A) is defined as follows:

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

where x, t, and r, c are, respectively, term, type and row variables. Correspondingly, the judgement are: $\Gamma \vdash *$, $\Gamma \vdash R \bullet C : \kappa$, $\Gamma \vdash \tau : T$ and $\Gamma \vdash e : \tau$. The judgement $\Gamma \vdash *$ can be read as "Γ is a well-formed context" and the meaning of the other judgements is the usual one.

3.2 Typing Rules for Objects

For the most part, the type system is routine. The object-related rules are discussed below. The first rule defines the type of the empty object: having no methods, the empty object needs no further method to be complete. Hence:

$$\frac{\Gamma \vdash *}{\Gamma \vdash \langle\rangle : \texttt{class}\, t.\langle\!\langle\rangle\!\rangle \bullet \langle\!\langle\rangle\!\rangle} \quad (\textit{empty object})$$

The typing rule to invoke messages has the format described in the previous subsection. The rule for method addition is defined as follows:

$$\frac{\begin{array}{ll} \Gamma \vdash \mathtt{e}_1 : \texttt{class}\, t.R \bullet \langle\!\langle C\lfloor n{:}\tau_\Delta, \mathbf{p{:}\gamma}\rfloor \rangle\!\rangle & \{\mathbf{m{:}\alpha}\} \in R \bullet C \\ \Gamma, t : T \vdash R \bullet C : [n, \mathbf{p}] & \Delta = \{\mathbf{m}, \mathbf{p}\} \\ \Gamma,\, r\bullet c : T \to [\mathbf{m}, n, \mathbf{p}] \vdash & \\ \quad \mathtt{e}_2 : [(\texttt{class}\, t.\langle\!\langle rt \mid \mathbf{m{:}\alpha}, n{:}\tau_\Delta, \mathbf{p{:}\gamma} \rangle\!\rangle \bullet ct)/t](t \to \tau) \;\; r, c \notin (\tau, \gamma) & \end{array}}{\Gamma \vdash \langle e_1 \longleftrightarrow\!\!\!\!\!+\; n{=}e_2\rangle : \texttt{class}\, t.\langle\!\langle R \mid n{:}\tau_\Delta \rangle\!\rangle \bullet \langle\!\langle C \mid \mathbf{p{:}\gamma} \rangle\!\rangle} \quad (\textit{obj ext})$$

where $\{\mathbf{m{:}\alpha}\} \in R \bullet C$ indicates that the $\mathbf{m{:}\alpha}$ methods are contained in $R \bullet C$, and $R \bullet \langle\!\langle C\lfloor n{:}\tau_\Delta, \mathbf{p{:}\gamma}\rfloor \rangle\!\rangle$ indicates that the completion-row of the type of e_1 may or may not contain $n, \mathbf{p}$. Whether or not n and the $\mathbf{p}$ methods must be included in the completion of the type of e_1 depends on whether or not the methods in e_1 contain n in the labels associated with their types. As for the n method being added, the set of its dependences may, in general, include the methods that are already contained in the object as well as methods that are yet to be added: the former are a subset of the $\mathbf{m}$ methods occurring in R, whereas the latter are the subset of $\mathbf{m}$ occurring in C together with the $\mathbf{p}$ (that are methods referenced to by n only). Note that all of the dependences of n are assumed to be part of the interface-row in the type of e_2: this, together with the condition $\Delta = \{\mathbf{m}, \mathbf{p}\}$, either checks (if n belongs to the type of e_1) – or otherwise it enforces – the constraint that Δ contains all methods that are referenced to (either directly, or indirectly) by e_2. To see this, consider the case when $e_2 \stackrel{def}{=} \lambda self.(self \Leftarrow p)$ 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, $\mathbf{q}$ methods in the label of the type of p. But then Δ, the label of n, must include p, a direct reference, as well as the $\mathbf{q}$ methods that n references indirectly via p.

Note, finally, that as in [5], the type of n has the form $t \to \tau$ (with a class type substituted for t) to conform with the self-application semantics of method invocation. Here, however, this type is polymorphic both in r and in c, and so that e_2 will have the indicated type for every R and C provided that R and C have the correct kind. Hence, invocations of n on future objects derived from $\langle e_1 \longleftrightarrow\!\!\!\!\!+\; n{=}e_2\rangle$ will be well-typed just in case these objects are complete with respect to n, i.e. they contain all of the methods upon which n depends.

The rule for method override is similar but simpler (see Appendix A): as for (*obj-ext*), the side-conditions on the labeled type of the method being overridden enforce the correct propagation of transitive references within labels. This may be observed as in the example above, taking $e_2 \stackrel{def}{=} \lambda self.\langle self \leftarrow p{=}\lambda s.(s \Leftarrow q)\rangle$ where q is, say, a constant method (whose type has an empty label).

3.3 Examples of Type Derivations

Example 1. This example shows that methods can be added in any order, regardless of the interdependences between them. Consider the following expression from [5]:

$$\mathtt{pt} \stackrel{def}{=} \langle\langle\langle\rangle \leftarrow\!\!+\ \mathtt{plus1} = \lambda\mathtt{s}.(\mathtt{s} \Leftarrow \mathtt{x}) + 1\rangle \leftarrow\!\!+\ \mathtt{x} = \lambda\mathtt{self}.3\rangle.$$

The above object cannot be typed with the system of [5] because the subexpression $\langle\langle\rangle \leftarrow\!\!+\ \mathtt{plus1} = \lambda\mathtt{s}.(\mathtt{s} \Leftarrow \mathtt{x}) + 1\rangle$ is not well-typed. Using the typing rule introduced in the previous section, instead, we may proceed as follows. Let $\Gamma_1 = r \bullet c : T{\to}[\mathtt{x}, \mathtt{plus1}],\ \mathtt{s} : \mathtt{class}\, t.\langle\!\langle rt \mid \mathtt{plus1}{:}int_{\{\mathtt{x}\}}, \mathtt{x}{:}int\rangle\!\rangle{\bullet} ct$. It is now easy to see that the following judgements are all derivable:

$$\begin{array}{rl}
\varepsilon \vdash & \langle\rangle : \mathtt{class}\, t.\langle\!\langle\rangle\!\rangle \bullet \langle\!\langle\rangle\!\rangle \\
\Gamma_1 \vdash & (\mathtt{s} \Leftarrow \mathtt{x}) + 1 : int, \\
\Gamma_1 - \mathtt{s} \vdash & \lambda\mathtt{s}.(\mathtt{s} \Leftarrow \mathtt{x}) + 1 : \mathtt{class}\, t.\langle\!\langle rt \mid \mathtt{plus1}{:}int_{\{\mathtt{x}\}}, \mathtt{x}{:}int\rangle\!\rangle \bullet ct{\to}int, \\
\varepsilon \vdash & \langle\langle\rangle \leftarrow\!\!+\ \mathtt{plus1} = \lambda\mathtt{s}.(\mathtt{s} \Leftarrow \mathtt{x}) + 1\rangle : \mathtt{class}\, t.\langle\!\langle \mathtt{plus1}{:}int_{\{\mathtt{x}\}}\rangle\!\rangle \bullet \langle\!\langle \mathtt{x}{:}int\rangle\!\rangle.
\end{array}$$

where the occurrence of *int* in the completion-row stands for $int_{\{\}}$. Now, letting $\Gamma_2 = r \bullet c : T{\to}[\mathtt{x}, \mathtt{plus1}],\ \mathtt{self} : \mathtt{class}\, t.\langle\!\langle rt \mid \mathtt{plus1}{:}int_{\{\mathtt{x}\}}, \mathtt{x}{:}int\rangle\!\rangle \bullet ct$, with a sequence of steps similar to the previous one, we may conclude with:

$$\varepsilon \vdash \mathtt{pt} : \mathtt{class}\, t.\langle\!\langle \mathtt{plus1}{:}int_{\{\mathtt{x}\}}, \mathtt{x}{:}int\rangle\!\rangle \bullet \langle\!\langle\rangle\!\rangle.$$

Clearly, the same typing discipline also allows us to add mutually recursive methods with no need to resort to the use of dummy methods as in [5].

Example 2. The last example motivates the requirement that the label associated with a method type contains both direct and indirect dependences for that method. Consider extending the `pt` object of the previous example as shown below:

$$\mathtt{newpt} \stackrel{def}{=} \langle \mathtt{p} \leftarrow\!\!+\ \mathtt{plus} = \lambda\mathtt{s}.\mathtt{s} \Leftarrow \mathtt{plus1}\rangle.$$

It can be verified that the following judgement may be derived:

$$\varepsilon \vdash \mathtt{newpt} : \mathtt{class}\, t.\langle\!\langle \mathtt{plus}{:}int_{\{\mathtt{plus1},\mathtt{x}\}}, \mathtt{plus1}{:}int_{\{\mathtt{x}\}}\rangle\!\rangle \bullet \langle\!\langle \mathtt{x}{:}int\rangle\!\rangle.$$

The point to notice is that the typing rules force the label of `plus` to include the `x` method, although `plus` depends on `x` indirectly via `plus1`. The reason why indirect references in the labels are need should now be clear: having only `plus1` in the label of `plus`, we would be able to type the invocation `newpt` $\Leftarrow$ `plus` which, instead, causes a *message not understood* error because `newpt` does not contain `x`.

4 Soundness of the Type System

The soundness of the type system is proved following the same schema as in [5]. We first show that types are preserved by the reduction process. Then we introduce an evaluation strategy that allows us to formalize the notion of error, and we show that that errors are detected statically by the type system.

To prove subject reduction, we need the following results that help isolate some interesting properties of the type system (proofs are omitted for the lack of space). Lemma 3 is used to specialize class-types to contain additional methods.

Lemma 3. *If* $\Gamma, r \bullet c : T^n \to [\mathbf{m}], \Gamma' \vdash e : \tau$ *and* $\Gamma \vdash R \bullet C : T^n \to [\mathbf{m}]$ *are both derivable, then so is* $\Gamma, [R/r, C/c]\Gamma' \vdash e : [R/r, C/c]\tau$.

The next two lemmas are used for building well–formed row expressions that can be substituted for row variables in typing derivations.

Lemma 4. *If* $\Gamma \vdash \mathtt{class}\, t.\langle\!\langle R \mid \mathbf{m}{:}\boldsymbol{\alpha}\rangle\!\rangle \bullet \langle\!\langle C \mid \mathbf{p}{:}\boldsymbol{\gamma}\rangle\!\rangle : T$ *is derivable, then so are* $\Gamma, t : T \vdash \alpha_i : T$ *for each* α_i *in* $\boldsymbol{\alpha}$, *and* $\Gamma, t : T \vdash \gamma_i : T$ *for each* γ_i *in* $\boldsymbol{\gamma}$, *and* $\Gamma, t : T \vdash R \bullet C : [\mathbf{m}, \mathbf{p}]$.

Lemma 5. *If* $\Gamma \vdash e : \tau$ *is derivable, then so is* $\Gamma \vdash \tau : T$.

Besides these results, in the following we will also assume that our type derivations be in normal form (the reader is referred to [3] for the definition of such normal form).

Theorem 6 ((Subject Reduction)). *If* $\Gamma \vdash e : \tau$ *is derivable, and* $e \overset{ev}{\twoheadrightarrow} e'$, *then* $\Gamma \vdash e' : \tau$ *is also derivable.*

Proof. The proof is by induction on the number of steps in $e \overset{ev}{\twoheadrightarrow} e'$. For the basic step (i.e. one $\overset{ev}{\to}$ step), we proceed by cases on the definition of $\overset{ev}{\to}$ and use induction on the context of the redex. The proof for (β) is standard, whereas for (*perm*) the proof is by induction on the structure of the first judgement: it distinguishes the four cases that arise when each occurrence of $\hookleftarrow\!\circ$ is either $\longleftarrow\!\!+$ or $\leftarrow$ and, in each such case, it further distinguishes four sub-cases according to the possible mutual dependences between n and m.

The remaining case is $(\Leftarrow)$: what we need to show is that if $\Gamma \vdash \langle e \hookleftarrow\!\circ\; n{=}e_n\rangle \Leftarrow n : \tau$ is derivable, then so is $\Gamma \vdash e_n\langle e \hookleftarrow\!\circ\; n{=}e_n\rangle : \tau$. Again, we need to distinguish the possible instances of the $\hookleftarrow\!\circ$ operator: below we consider the case when $\hookleftarrow\!\circ$ is $\longleftarrow\!\!+$, the case when $\hookleftarrow\!\circ$ is $\leftarrow$ being similar. The proof is by induction on the derivation of $\Gamma \vdash \langle e \hookleftarrow\!\circ\; n{=}e_n\rangle \Leftarrow n : \tau$. For rule (*weak*) it follows directly from the induction hypothesis, so we consider only rule (*send*).

If the last applied rule is (*send*), then the derivation has the following form:

$$\dfrac{\dfrac{\Xi}{\Gamma \vdash \langle e \longleftarrow\!\!+\; n{=}e_n\rangle : \mathtt{class}\, t.\langle\!\langle R \mid \mathbf{m}{:}\boldsymbol{\alpha}, n{:}\tau_{\{\mathbf{m}\}}\rangle\!\rangle \bullet C}}{\Gamma \vdash \langle e \longleftarrow\!\!+\; n{=}e_n\rangle \Leftarrow n : [(\mathtt{class}\, t.\langle\!\langle R \mid \mathbf{m}{:}\boldsymbol{\alpha}, n{:}\tau_{\{\mathbf{m}\}}\rangle\!\rangle \bullet C)/t]\tau} \quad (send)$$

The interesting case is when Ξ ends up with (*obj ext*), the only other possibility being (*weak*). Note, further, that the (*obj ext*) may only have the form:

$$\frac{\begin{array}{l}\Gamma \vdash e : \mathtt{class}\, t.\langle\!\langle R \mid \mathbf{m{:}\alpha} \rangle\!\rangle \bullet \langle\!\langle C \lfloor n{:}\tau_{\{\mathbf{m}\}} \rfloor \rangle\!\rangle \\ \Gamma, t : T \vdash \langle\!\langle R \mid \mathbf{m{:}\alpha} \rangle\!\rangle \bullet C : [n] \\ \Gamma, r \bullet c : T \to [\mathbf{m}, n] \vdash e_n : [(\mathtt{class}\, t.\langle\!\langle rt \mid \mathbf{m{:}\alpha}, n{:}\tau_{\{\mathbf{m}\}} \rangle\!\rangle \bullet ct)/t](t \to \tau)\end{array}}{\Gamma \vdash \langle e \leftarrow\!\!+\, n{=}e_n \rangle : \mathtt{class}\, t.\langle\!\langle R \mid \mathbf{m{:}\alpha}, n{:}\tau_{\{\mathbf{m}\}} \rangle\!\rangle \bullet C} \quad (obj\ ext)$$

Now we show that a derivation exists for $\Gamma \vdash e_n \langle e \leftarrow\!\!+\, n{=}e_n \rangle : [(\mathtt{class}\, t.\langle\!\langle R \mid \mathbf{m{:}\alpha}, n{:}\tau_{\{\mathbf{m}\}} \rangle\!\rangle \bullet C)/t]\tau$ regardless of whether n is in the type of e_1 or not. First note that the judgment: $\Gamma, t : T \vdash R \bullet C : [\mathbf{m}, n]$ is derivable. If n is in the type of e, from $\Gamma \vdash e : \mathtt{class}\, t.\langle\!\langle R \mid \mathbf{m{:}\alpha} \rangle\!\rangle \bullet \langle\!\langle C \mid n{:}\tau_{\{\mathbf{m}\}} \rangle\!\rangle$, by Lemma 5, we have that $\Gamma \vdash \mathtt{class}\, t.\langle\!\langle R \mid \mathbf{m{:}\alpha} \rangle\!\rangle \bullet \langle\!\langle C \mid n{:}\tau_{\{\mathbf{m}\}} \rangle\!\rangle : T$ is derivable and the claim follows by Lemma 4. Otherwise, the claim derives directly from $\Gamma, t : T \vdash \langle\!\langle R \mid \mathbf{m{:}\alpha} \rangle\!\rangle \bullet C : [n]$. By an application of (*rabs*), we then derive $\Gamma \vdash \lambda t.R \bullet \lambda t.C : T \to [\mathbf{m}, n]$. From this, by applying Lemma 3 to the typing of e_n, we next derive the judgement: $\Gamma \vdash e_n : [(\mathtt{class}\, t.\langle\!\langle R \mid \mathbf{m{:}\alpha}, n{:}\tau_{\{\mathbf{m}\}} \rangle\!\rangle \bullet C)/t](t \to \tau)$. From this, an application of (*eapp*) completes the derivation of the desired judgement. □

4.1 Type Soundness

We conclude formalizing the notion of *message not understood* errors. Intuitively, an error occurs when a message n is sent to an object that does not contain n. The structural operational semantics that formalizes this situation is defined as in [5] in terms of two functions, *eval* and get_n: the *eval* function is a variation of the standard *lazy* evaluator for the λ-calculus that, when fed with an expression of the form $e \Leftarrow n$, calls the get_n function to extract a definition for the n method from e. To perform method extraction, get_n inspects e (possibly evaluating it) until it either finds a definition for the n method, or it determines that e does not contain any definition for n: in the first case get_n returns the body of n as a result, in the second it returns *error*. The complete set of evaluation rules is presented in Appendix B.

Due to the lack of space, we only give the statement of the soundness theorem (and of the main lemmas) and omit proofs. Proofs are carried out exactly as in [5] by induction on the definition of the *eval* and *get* functions, using Subject Reduction. As in that case, since the result of evaluation may be undefined besides being successful or error, it is easier to prove the contrapositive of soundness, i.e. that if $eval(e) = error$, then e cannot be typed.

Lemma 7. *If $ev(e) = e'$ then $e \stackrel{ev}{\twoheadrightarrow} e'$, where ev is either eval or get.*

Let the notation $\Gamma \nvdash A$ indicate that the judgement $\Gamma \vdash A$ is not derivable. Then the following holds:

Lemma 8. *i) If $get_n(e) = error$, then $\varepsilon \nvdash e : \mathtt{class}\, t.\langle\!\langle R \mid n{:}\alpha \rangle\!\rangle \bullet C$, for any R, C rows, and α labeled-type.*
ii) If $eval(e) = error$, then $\varepsilon \nvdash e : \tau$ for any type τ.

Theorem 9 ((Soundness)). *If $\varepsilon \vdash e : \tau$ is derivable, then $eval(e) \neq error$.*

5 Conclusions

We have presented an extension of the *Lambda Calculus of Objects* [5] with a new type system that supports a novel and more flexible typing discipline, while preserving all of the interesting properties of the original system. The new calculus enjoys a more elegant operational semantics and the additional expressive power that derives from the possibility of computing with objects whose implementation is only partially specified. The new features are accounted for by extending the class-types from [5] with two technical tools: *completion-rows* and *labeled-types*. Completion-rows convey information about methods yet to be added, thus allowing sequences of method additions to be typed regardless their order. Labeled-types, in turn, encode information on the structure of a method body that allows a method invocation to be type correctly even though the receiver of the message is an incomplete object. The new features induce only little additional complexity for the typing rules and, furthermore, as we show in [2], the use of labeled-types makes the type system amenable to a smooth integration with the notion of *width* subtyping of [3].

Acknowledgements We wish to thank Mariangiola Dezani-Ciancaglini for her insightful suggestions in countless discussions. Thanks are also due to Laurent Dami for pointing out a flaw in the preliminary version of this paper, and to the anonymous referees, whose remarks helped improve the technical presentation of the final version substantially.

References

1. M. Abadi and L. Cardelli. A Theory of Primitive Objects. In *Proceedings of Theoretical Aspect of Computer Software*, volume 789 of *LNCS*, pages 296–320. Springer-Verlag, 1994.
2. V. Bono, M. Bugliesi, and L. Liquori. A Calculus of Incomplete objects with Subtyping. In preparation.
3. V. Bono and L. Liquori. A Subtyping for the Fisher-Honsell-Mitchell Lambda Calculus of Objects. In *Proceedings of International Conference of Computer Science Logic*, volume 933 of *LNCS*, 1995.
4. E. Ellis and B. Stroustrop. *The Annotated C^{++} Reference Manual*. ACM Press, 1990.
5. 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.
6. K. Fisher and J. C. Mitchell. A Delegation-based Object Calculus with Subtyping. In *Proceedings of FCT-95*, Lecture Notes in Computer Science. Springer-Verlag, 1995. To appear.
7. A. Goldberg and D. Robson. *Smalltalk-80, The Language and its Implementation.* Addison Wesley, 1983.
8. J. C. Michell. Toward a Typed Foundation for Method Specialization and Inheritance. In *Proc. 17th ACM Symp. on Principles of Programming Languages*, pages 109–124. ACM, 1990.
9. D. Ungar and R. B. Smith. Self: the power of simplicity. In *Proceedings of ACM Symp. on Object-Oriented Programming Systems, Languages, and Applications*, pages 227–241. ACM Press, 1987.

A Typing Rules

General Rules ($a : b$ is either $e : \tau$, or $t : T$, or $r{\bullet}c : \kappa$)

$$\frac{}{\varepsilon \vdash *}\ (ax) \qquad \frac{\Gamma \vdash * \quad a : b \in \Gamma}{\Gamma \vdash a : b}\ (proj)$$

$$\frac{\Gamma \vdash * \quad a \notin Dom(\Gamma)}{\Gamma, a{:}b \vdash *}\ (var) \qquad \frac{\Gamma \vdash a : b \quad \Gamma, \Gamma' \vdash *}{\Gamma, \Gamma' \vdash a : b}\ (weak)$$

Rules for Types

$$\frac{\Gamma \vdash \tau_1 : T \quad \Gamma \vdash \tau_2 : T}{\Gamma \vdash \tau_1 \to \tau_2 : T}\ (t{-}app) \qquad \frac{\Gamma, t{:}T \vdash R{\bullet}C : [\mathbf{m}]}{\Gamma \vdash \mathtt{class}\, t.R{\bullet}C : T}\ (class)$$

Types and Row Equality

$$\frac{\Gamma \vdash \tau : T \quad \tau \to_\beta \tau'}{\Gamma \vdash \tau' : T}\ (t{-}\beta) \qquad \frac{\Gamma \vdash R{\bullet}C : \kappa \quad R{\bullet}C \to_\beta R'{\bullet}C'}{\Gamma \vdash R'{\bullet}C' : \kappa}\ (r{-}\beta)$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau' : T \quad \tau \leftrightarrow_\beta \tau'}{\Gamma \vdash e : \tau'}\ (t{-}eq)$$

Rules for Rows

$$\frac{\Gamma \vdash *}{\Gamma \vdash \langle\!\rangle{\bullet}\langle\!\rangle : [\mathbf{m}]}\ (er) \qquad \frac{\Gamma \vdash C{\bullet}R : \kappa}{\Gamma \vdash R{\bullet}C : \kappa}\ (rexc)$$

$$\frac{\Gamma \vdash R{\bullet}C : T^n \to [\mathbf{m}] \quad \{\mathbf{n}\} \subseteq \{\mathbf{m}\}}{\Gamma \vdash R{\bullet}C : T^n \to [\mathbf{n}]}\ (rlab) \qquad \frac{\begin{array}{l}\Gamma \vdash \tau : T \\ \Gamma \vdash R{\bullet}C : [\mathbf{m}, n]\ n \notin \Delta\end{array}}{\Gamma \vdash \langle R \mid n{:}\tau_\Delta \rangle{\bullet}C : [\mathbf{m}]}\ (rext)$$

$$\frac{\Gamma, t{:}T \vdash R{\bullet}C : T^n \to [\mathbf{m}]}{\Gamma \vdash \lambda t.R{\bullet}\lambda t.C : T^{n+1} \to [\mathbf{m}]}\ (rabs) \qquad \frac{\begin{array}{l}\Gamma \vdash \tau : T \\ \Gamma \vdash R{\bullet}C : T^{n+1} \to [\mathbf{m}]\end{array}}{\Gamma \vdash R\tau{\bullet}C\tau : T^n \to [\mathbf{m}]}\ (rapp)$$

Rules for Expressions

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}\ (eabs) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau_2}\ (eapp)$$

$$\frac{\Gamma \vdash *}{\Gamma \vdash \langle\rangle : \mathtt{class}\, t.\langle\!\rangle{\bullet}\langle\!\rangle}\ (\langle\rangle) \qquad \frac{\Gamma \vdash e : \mathtt{class}\, t.\langle R \mid \mathbf{n{:}\boldsymbol{\alpha}}, m{:}\tau_{\{\mathbf{n}\}} \rangle{\bullet}C}{\Gamma \vdash e \Leftarrow m : [(\mathtt{class}\, t.\langle R \mid \mathbf{n{:}\boldsymbol{\alpha}}, m{:}\tau_{\{\mathbf{n}\}} \rangle{\bullet}C)/t]\tau}\ (send)$$

$$\frac{\begin{array}{ll} \Gamma \vdash \mathtt{e}_1 : \mathtt{class}\, t.R \bullet \langle\!\langle C \lfloor n{:}\tau_\Delta, \mathbf{p}{:}\boldsymbol{\gamma} \rfloor \rangle\!\rangle & \{\mathbf{m}{:}\boldsymbol{\alpha}\} \in R \bullet C \\ \Gamma, t : T \vdash R \bullet C : [n, \mathbf{p}] & \Delta = \{\mathbf{m}, \mathbf{p}\} \\ \Gamma,\ r{\bullet}c : T \to [\mathbf{m}, n, \mathbf{p}] \vdash & \\ \quad \mathtt{e}_2 : [(\mathtt{class}\, t.\langle\!\langle rt \mid \mathbf{m}{:}\boldsymbol{\alpha}, n{:}\tau_\Delta, \mathbf{p}{:}\boldsymbol{\gamma} \rangle\!\rangle{\bullet}ct)/t](t \to \tau) & r, c \notin (\tau, \boldsymbol{\gamma}) \end{array}}{\Gamma \vdash \langle e_1 \leftarrow\!\!+\ n{=}e_2 \rangle : \mathtt{class}\, t.\langle\!\langle R \mid n{:}\tau_\Delta \rangle\!\rangle{\bullet}\langle\!\langle C \mid \mathbf{p}{:}\boldsymbol{\gamma} \rangle\!\rangle} \quad (\mathit{obj\ ext})$$

$$\frac{\begin{array}{ll} \Gamma \vdash \mathtt{e}_1 : \mathtt{class}\, t.\langle\!\langle R \mid n{:}\tau_\Delta \rangle\!\rangle{\bullet}C & \{\mathbf{m}{:}\boldsymbol{\alpha}\} \in R \bullet C \\ \Gamma, t : T \vdash R \bullet C : [n] & \Delta = \{\mathbf{m}\} \\ \Gamma,\ r{\bullet}c : T \to [\mathbf{m}, n] \vdash & \\ \qquad \mathtt{e}_2 : [(\mathtt{class}\, t.\langle\!\langle rt \mid \mathbf{m}{:}\boldsymbol{\alpha}, n{:}\tau_\Delta \rangle\!\rangle{\bullet}ct)/t](t \to \tau) & \end{array}}{\Gamma \vdash \langle e_1 \leftarrow\ n{=}e_2 \rangle : \mathtt{class}\, t.\langle\!\langle R \mid n{:}\tau_\Delta \rangle\!\rangle{\bullet}C} \quad (\mathit{obj\ over})$$

B Evaluation Strategy

Inference rules for get (z is either an expression or err)

$$\frac{}{get_n(x) = err} \quad (get_n\ var) \qquad \frac{}{get_n(\langle\rangle) = err} \quad (get_n\ \langle\rangle)$$

$$\frac{}{get_n(\lambda x.e) = err} \quad (get_n\ abs)$$

$$\frac{}{get_n(\langle e_1 \leftarrow\!\circ\ n{=}e_2 \rangle) = e_2} \quad (get_n\ succ) \qquad \frac{get_n(e_1) = z \quad (m \neq n)}{get_n(\langle e_1 \leftarrow\!\circ\ m{=}e_2 \rangle) = z} \quad (get_n\ next)$$

$$\frac{\begin{array}{l} get_n(e) = e_1 \\ get_n(e_1\, e) = z \end{array}}{get_n(e \Leftarrow n) = z} \quad (get_n \Leftarrow) \qquad \frac{get_n(e) = err}{get_n(e \Leftarrow n) = err} \quad (get_n \Leftarrow\ err)$$

$$\frac{\begin{array}{l} eval(e_1) = \lambda x.e \\ get_n([e_2/x]\, e) = z \end{array}}{get_n(e_1\, e_2) = z} \quad (get_n\ app) \qquad \frac{eval(e_1) = err}{get_n(e_1\, e_2) = err} \quad (get_n\ app\ err)$$

Inference rules for eval

$$\frac{}{ev(x) = x} \quad (eval\ var) \qquad \frac{}{ev(\langle\rangle) = \langle\rangle} \quad (eval\ \langle\rangle) \qquad \frac{}{eval(\lambda x.e) = \lambda x.e} \quad (eval\ abs)$$

$$\frac{}{eval(\langle e_1 \leftarrow\!\circ\ m{=}e_2 \rangle) = \langle e_1 \leftarrow\!\circ\ m{=}e_2 \rangle} \quad (eval\ obj)$$

$$\frac{\begin{array}{l} get_n(e) = e_1 \\ eval(e_1 e) = z \end{array}}{eval(e \Leftarrow n) = z} \quad (eval\ \Leftarrow) \qquad \frac{get_n(e) = err}{eval(e \Leftarrow n) = err} \quad (eval\ \Leftarrow\ err)$$

$$\frac{\begin{array}{l} eval(e_1) = \lambda x.e \\ eval([e_2/x]e) = z \end{array}}{eval(e_1 e_2) = z} \quad (eval\ app) \qquad \frac{eval(e_1) = err}{eval(e_1 e_2) = err} \quad (eval\ app\ err)$$