Matching Constraints for the Lambda Calculus of Objects

Matching Constraints for the Lambda Calculus of Objects

Viviana Bono
Dipartimento di Informatica, Università di Torino
C.so Svizzera 185, I-10149 Torino, Italy
e-mail: bono@di.unito.it

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

Abstract. We present a new type system for the *Lambda Calculus of Objects* [16], based on matching. The new system retains the property of type safety of the original system, while using implicit match-bounded quantification over type variables instead of implicit quantification over row schemes (as in [16]) to capture *Mytype* polymorphic types for methods. Type soundness for the new system is proved as a direct corollary of subject reduction. A study of the relative expressive power of the two systems is also carried out, that shows that the new system is as powerful as the original one on derivations of closed-object typing judgements. Finally, an extension of the new system is presented, that gives provision for a class-based calculus, where primitives such as creation of class instances and method update are rendered in terms of delegation.

1 Introduction

The problem of deriving safe and flexible type systems for object-oriented languages has been addressed by many theoretical studies in the last years. The interest of these studies has initially been focused on *class-based* languages, while, more recently, type systems have also been proposed for *object-based* (or *delegation-based*) languages. Clearly, the work on the latter has been strongly influenced by the study of the former. For example, the notion of *row-variables* introduced by [18] to type extensible records was refined in [16, 7, 17, 5] to type extensible objects. Similarly, *recursive object types* have first been used to provide functional models of class-based languages [11, 9, 12, 14, 13], and then applied to the case of an object calculus supporting method override in presence of subsumption [3]. A further notion that has been studied extensively in class-based languages (as well as in the record calculus of [10]) is that of *(F-)bounded quantification* as a tool for modeling the subclass relation. Following this line of research, in this paper we investigate the role of *matching* in the design of a type system for the delegation-based *Lambda Calculus of Objects* [16].

Matching is a relation over object types that has first been introduced by [8] as an alternative to F-bounded subtyping [13] in modeling the subclass relation in class-based languages, and then as a complement to subtyping to model method inheritance between classes [1, 2].

The *Lambda Calculus of Objects* [16] is a delegation-based calculus where method addition and override, as well as method inheritance, all take place at the object level rather than at the class level. In [16] 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 system we present in this paper takes a different approach to the rendering of *Mytype* specialization, while retaining the property of type safety of the original system. Technically, the new solution is based on implicit match-bounded quantification over type variables to characterize methods as functions with polymorphic types, and to enfore correct instantiation of these types as methods are inherited.

A similar solution for the polymorphic typing of methods in Lambda Calculus of Objects is proposed in [6]. The key difference, with respect to the system of this paper, is that [6] uses subtyping, instead of matching, and (implicit) subtype-bounded quantification. There appear to be fundamental tradeoffs between the two solutions: in fact, while subtyping has the advantage of allowing object subsumption, matching appears to be superior to subtyping in the rendering of the desired typing of methods. The reason is explained, briefly, as follows: in order to ensure safe uses of subsumption, the system of [6] allows type promotion for an object-type only when the methods in the promoted type do not reference any of the methods of the original type. As in [7], labeled types are used to encode the cross references among methods in the methods' types. In [6], however, labeled types involve some additional limitations over [7] for subsumption, and require a rather more complex bookkeeping that affects the typing of methods as well.

Relying on matching, instead, has the advantage of isolating the typing of methods from subtyping, thus allowing a rather elegant and simple rendering of the method polymorphism. The simplicity of the resulting system also allows us to draw a formal analysis of the relationship between the original system and ours. In particular, we show that every closed-object typing judgement derivable in [16] is also derivable in our system. We then show that the new system may naturally be extended to give provision for a class-based calculus, where subclassing, and primitives such as creation of class instances and method update are rendered in terms of delegation.

The rest of the paper is organized as follows. In Section 2, we review the untyped calculus of [16]. In Section 3, we present the new typing rules for objects and we prove type soundness. In Section 4, we present the encoding of the type system of [16] into the new system. In Section 5, we present the extended system that gives provision for classes, and then we conclude in Section 6 with some final remarks.

2 The Untyped Calculus

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

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

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

$\langle\rangle$	is the empty object,
$\langle e_1 \leftarrow\!\!+\, m{=}e_2\rangle$	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,
$e \Leftarrow m$	sends message m to object e,

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. As in [16], both these conditions are enforced statically by the type system.

The other main object operation is method invocation, which comprises two separate actions, *search* and *self-application*: evaluating the message $e \Leftarrow m$ requires a search, within e, of the body of the m method which is then applied to e itself. To formalize this behavior, we introduce a subsidiary object expression, $e \hookleftarrow m$, whose intuitive semantics is as follows: evaluating $e \hookleftarrow m$ results into a recursive traversal of the "sub-objects" of e, that succeeds upon reaching the right-most addition or override of the method in question.

The operational semantics $\xrightarrow{eval}$ of the untyped calculus can be thus defined as the reflexive, transitive and contextual closure of the reduction relation defined below ($\leftarrow\!\!\circ$ stands for both $\leftarrow\!\!+$ and $\leftarrow$)

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

plus a few additional reductions for error propagation (see Appendix A). The use of the search operator expressions in our calculus is inspired to [7], and it provides a more direct and concise technical device than the *bookkeeping* relation originally introduced in [16].

3 Object Types and Matching

Object types have the same structure as in [16]: an object type has the form

$$\texttt{Obj}\, t.\langle\!\langle m_1{:}\tau_1, \ldots, m_k{:}\tau_k \rangle\!\rangle$$

where the m_i's are method names, whereas the τ_i's are type expressions.

The row $\langle\!\langle m_1{:}\tau_1, \ldots, m_k{:}\tau_k \rangle\!\rangle$ defines the interface or *protocol* of the objects of that type, i.e. the list of the methods (with their types) that may be invoked on these objects. The binder `Obj` scopes over the row, and the bound variable t may occur free within the scope of the binder, with every free occurrence referring to the object type itself. `Obj`-types are thus a form of recursively-defined types, even though `Obj` is not to be understood as a fixed-point operator: as in [16], the self-referential nature of these types is axiomatized directly by the typing rules, rather than defined in terms of an explicit unfolding rule.

3.1 Types and Rows

Type expressions include type-constants, type variables, function types and object types. The sets of rows and types are defined recursively as follows:

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

The symbol b denotes type constants, t, u, and v denote type variables, whereas τ, σ, ρ, ... range over types; all symbols may appear indexed.

Row expressions that differ only for the name of the bound variable, or for the order of the component $m{:}\tau$ pairs are considered identical. More formally, α-conversion of type variables bound by `Obj`, as well as applications of the principle:

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

are taken as syntactic conventions rather than as explicit rules. The structure of valid contexts (see Appendix B) is defined as follows:

$$\Gamma ::= \varepsilon \mid \Gamma,\, x : \tau \mid \Gamma,\, u \lessdot\!\!\# \ \tau,$$

Correspondingly, the judgements are $\Gamma \vdash *$, $\Gamma \vdash e : \tau$, $\Gamma \vdash \tau_1 \lessdot\!\!\# \ \tau_2$, where $\Gamma \vdash *$ stands for "Γ is a well-formed context", and the reading of the other judgements is standard.

As an important remark, we note that, as in [6] and in contrast to [16], rows in our system are formed only as "ground" collections of pairs "method-name:type". One advantage of this choice is a simplified notion of well-formedness for rows: instead of the kinding judgements of [16], in our system this notion is axiomatized, syntactically, as follows. Let $\mathcal{M}(R)$ denote the set of method names of the row R defined inductively as follows:

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

Then, we say that a row is well formed if and only if it is (i) either the empty row $\langle\!\langle\rangle\!\rangle$, or (ii) a row of the form $\langle\!\langle R \mid m{:}\tau \rangle\!\rangle$ with R well formed and $m \notin \mathcal{M}(R)$.

3.2 Matching

Matching is the only relation over types that we assume in the type system; it is a reflexive and transitive relations over all types, while for `Obj`-types it also formalizes the notion of *protocol extension* needed in the typing of inheritance. The relation we use here is a specialization of the original matching relation [8], defined by the following rule ($\overline{m{:}\tau}$ is short for $m_1{:}\tau_1, \ldots, m_k{:}\tau_k$):

$$\frac{\Gamma \vdash *\qquad \langle\!\langle \overline{m{:}\tau}, \overline{n{:}\sigma} \rangle\!\rangle \ \mathit{well\ formed}}{\Gamma \vdash \mathtt{Obj}\, t.\langle\!\langle \overline{m{:}\tau}, \overline{n{:}\sigma} \rangle\!\rangle \lessdot\!\!\# \ \mathtt{Obj}\, t.\langle\!\langle \overline{m{:}\tau} \rangle\!\rangle} \quad (\lessdot\!\!\#)$$

Unlike the original definition [8], that allows the component types of a `Obj`-type to be promoted by subtyping, our definition requires that these types coincide with the component types of every `Obj`-type placed higher-up in the $\lessdot\!\!\#$ -hierarchy. Like the original relation, on the other hand, our relation has the peculiarity that it is *not* used in conjunction with a subsumption rule. As noted in [2], this restriction is crucial to prevent unsound uses of type promotion in the presence of method override.

3.3 Typing Rules for Objects

For the most part, the type system is routine. The object-related rules are discussed below. The first defines the type of the empty object (whose type is the top of object-types in the $\lessdot\!\!\#$ -hierarchy):

$$\frac{\Gamma \vdash *}{\Gamma \vdash \langle\rangle : \mathtt{Obj}\, t.\langle\!\langle\rangle\!\rangle} \quad (\mathit{empty\ object}).$$

The typing rule for method invocation has the following format:

$$\frac{\Gamma \vdash e : \sigma \quad \Gamma \vdash \sigma \lessdot\!\!\# \ \mathtt{Obj}\, t.\langle\!\langle n{:}\tau \rangle\!\rangle}{\Gamma \vdash e \Leftarrow n : [\sigma/t]\tau} \quad (\mathit{send}).$$

As in [16], the substitution for t in τ reflects the recursive nature of object types. In order for a call to an n method on an object e to be typed, we require that e has any type containing the method name n. An interesting aspect of the above rule is that the type σ may either be a `Obj`-type matching $\mathtt{Obj}\, t.\langle\!\langle n{:}\tau \rangle\!\rangle$, or else an unknown type (i.e. a type variable) occurring (match-bounded) in the context Γ. Rules like (*send*) are sometimes referred to as *structural rules* [1], and their use is critical for an adequate rendering of *Mytype* polymorphism: it is the ability to refer to possibly unknown types in the type rules, in fact, that allows methods to act parametrically over any $u \lessdot\!\!\# A$, where u is the type of *self*, and A is a given `Obj`-type.

The next rule defines the typing of an object-extension with a new method:

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \mathtt{Obj}\, t.\langle\!\langle R \mid \overline{m{:}\rho} \rangle\!\rangle \\ \Gamma,\ u \lessdot\!\!\# \ \mathtt{Obj}\, t.\langle\!\langle \overline{m{:}\rho}, n{:}\tau \rangle\!\rangle \vdash e_2 : [u/t](t{\to}\tau) \qquad n \notin \mathcal{M}(\langle\!\langle R \mid \overline{m{:}\tau} \rangle\!\rangle)\end{array}}{\Gamma \vdash \langle e_1 \longleftarrow n{=}e_2 \rangle : \mathtt{Obj}\, t.\langle\!\langle R \mid \overline{m{:}\rho}, n{:}\tau \rangle\!\rangle} \quad (\mathit{ext}).$$

Typing the method addition for n requires (i) that the n method be not in the type of the object e_1 that is being extended, and (ii) that the protocol of the resulting object contains at least the n method as well as the $\overline{m}$ methods needed to type the body e_2 of n.

The typing of a method override is defined similarly. As for the (*send*) rule, the generality that derives from the use of the type σ is needed to carry out derivations where the (*over*) rule is applied with e_1 variable (e.g. *self*).

$$\frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma \vdash \sigma \lessdot\!\!\# \ \mathtt{Obj}\, t.\langle\!\langle \overline{m{:}\rho}, n{:}\tau \rangle\!\rangle \qquad \Gamma,\, u \lessdot\!\!\# \ \mathtt{Obj}\, t.\langle\!\langle \overline{m{:}\rho}, n{:}\tau \rangle\!\rangle \vdash e_2 : [u/t](t{\to}\tau)}{\Gamma \vdash \langle e_1 \leftarrow n{=}e_2 \rangle : \sigma} \quad (over).$$

We conclude with the rule for typing a search expression:

$$\frac{\Gamma \vdash e : \sigma \quad \Gamma \vdash \sigma \lessdot\!\!\# \ \mathtt{Obj}\, t.\langle\!\langle n{:}\tau \rangle\!\rangle \quad \Gamma \vdash \varsigma \lessdot\!\!\# \ \sigma}{\Gamma \vdash e \hookleftarrow n : [\varsigma/t](t{\to}\tau)} \quad (search).$$

Once more we assume a possibly unknown type for e: typing a search requires, however, more generality than typing a method invocation because the search of a method encompasses a recursive inspection of the recipient object (i.e. of *self*). This explains the intuitive roles of the two types ς and σ in the above rule: ς is the type of the *self* object, to which the n message was sent and to which the body of n will be applied; σ, on the other hand, is the type of e, the sub-object of *self* where the body of n method is eventually found while searching within *self*.

3.4 An Example of Type Derivations

We conclude the description of the type system with an example that illustrates the use of the typing rules in typing derivations. The example is borrowed from [16], and shows that the type system captures the desired form of method specialization. The following object expression represents a point object with an `x` coordinate and a `move` method:

$$\mathtt{pt} \equiv \langle \langle \mathtt{x} = \lambda \mathtt{self}.3 \rangle \longleftarrow\!\!+ \ \mathtt{move} = \lambda \mathtt{self}.\lambda \mathtt{dx}.\langle \mathtt{self} \leftarrow \mathtt{x} = \lambda \mathtt{s}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx} \rangle \rangle.$$

Below, we sketch the derivation of the judgement $\varepsilon \vdash \mathtt{pt} : \mathtt{Obj}\, t.\langle\!\langle \mathtt{x} : \mathit{int}, \mathtt{move} : \mathit{int}{\to}t \rangle\!\rangle$, using the assumption $\varepsilon \vdash \langle \mathtt{x} = \lambda \mathtt{self}.3 \rangle : \mathtt{Obj}\, t.\langle\!\langle \mathtt{x} : \mathit{int} \rangle\!\rangle$.
Consider then defining `cp` as a new point, obtained from `pt` with the addition of a `color` method, namely: $\mathtt{cp} \equiv \langle \mathtt{pt} \longleftarrow\!\!+ \ \mathtt{color} = \lambda \mathtt{self}.\mathtt{blue} \rangle$. With a similar derivation, we may now prove that `cp` has type $\mathtt{Obj}\, t.\langle\!\langle \mathtt{x} : \mathit{int}, \mathtt{move} : \mathit{int}{\to}t, \mathtt{color} : \mathit{colors} \rangle\!\rangle$, thus showing how the type of `move` gets specialized as the method is inherited from a `pt` to a `cp`.

Contexts

$\Gamma_1 \;=\; u \lessdot\!\!\# \; \mathtt{Obj}\, t.\langle \mathtt{x} : int, \mathtt{move} : int{\to}t\rangle,\ \mathtt{self} : u,\ \mathtt{dx} : int$

$\Gamma_2 \;=\; \Gamma_1,\ v \lessdot\!\!\# \; \mathtt{Obj}\, t.\langle \mathtt{x} : int, \mathtt{move} : int{\to}t\rangle,\ \mathtt{s} : v$

Derivation

1. $\Gamma_2 \vdash (\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx} : int$
   by (*send*) from $\Gamma_2 \vdash \mathtt{self} : u$ and $\Gamma_2 \vdash u \lessdot\!\!\# \; \mathtt{Obj}\, t.\langle \mathtt{x} : int\rangle$.
2. $\Gamma_2 - \mathtt{s} \vdash \lambda \mathtt{s}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx} : v{\to}int$
3. $\Gamma_1 \vdash \langle \mathtt{self} \leftarrow \mathtt{x} = \lambda \mathtt{s}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx}\rangle : u$
   by (*over*) from $\Gamma_1 \vdash \mathtt{self} : u$ and $\Gamma_1 \vdash u \lessdot\!\!\# \; \mathtt{Obj}\, t.\langle \mathtt{x} : int, \mathtt{move} : int{\to}t\rangle$ and 2.
4. $\Gamma_1 - \mathtt{self} - \mathtt{dx} \vdash \lambda \mathtt{self}.\lambda \mathtt{dx}.\langle \mathtt{self} \leftarrow \mathtt{x} = \lambda \mathtt{s}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx}\rangle : u{\to}int{\to}u$
5. $\varepsilon \vdash \mathtt{pt} : \mathtt{Obj}\, t.\langle \mathtt{x} : int, \mathtt{move} : int{\to}t\rangle$
   by (*ext*) from $\varepsilon \vdash \langle \mathtt{x} = \lambda \mathtt{self}.3\rangle : \mathtt{Obj}\, t.\langle \mathtt{x} : int\rangle$ and 4.

3.5 Type Soundness

We conclude this section with a theorem of type soundness. We first show that types are preserved by reduction.

Theorem 1 [**Subject Reduction**]. *If* $e_1 \xrightarrow{eval}\!\!\!\!\!\rightarrow\; e_2$ *and the judgement* $\Gamma \vdash e_1 : \tau$ *is derivable, then so is the judgement* $\Gamma \vdash e_2 : \tau$.

The proof is omitted here due to the lack of space, and can be found in [4]. Type soundness follows directly as a corollary of Subject Reduction, for if we may derive a type for an expression e, then e may not be reduced to *err* (which has no type). Hence we have:

Theorem 2 [**Type Soundness**]. *If* $\varepsilon \vdash e : \tau$ *is derivable, then* $e \not\xrightarrow{eval}\!\!\!\!\!\rightarrow\; wrong$.

4 Relationship with the Type System of [FHM94]

The relationship between our system and the system of [16] is best illustrated by looking at the example of Section 3.4, where we showed that the `move` method for the `pt` object may be typed with the following judgement:

$$u \lessdot\!\!\# \; \mathtt{Obj}\, t.\langle \mathtt{x} : int, \mathtt{move} : int{\to}t\rangle \vdash \lambda \mathtt{self}.\lambda \mathtt{x}.\langle \mathtt{self} \leftarrow \mathtt{x} = \lambda \mathtt{s}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx}\rangle : u{\to}int{\to}u$$

The corresponding judgement in the original system, (cfr. [16], Section 3.4)) is:

$$r : T{\to}[\mathtt{x}, \mathtt{move}] \vdash \lambda \mathtt{self}.\lambda \mathtt{x}.\langle \mathtt{self} \leftarrow \mathtt{x} = \lambda \mathtt{s}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx}\rangle : [\mathtt{Obj}\, t.\langle\!\langle rt \mid \mathtt{x} : int, \mathtt{move} : int{\to}t\rangle\!\rangle / t](t{\to}int{\to}t).$$

The two judgements have essentially the same structure, but the rendering of polymorphism is fundamentally different. In the original system, polymorphism is placed *inside* the row of the type $\mathtt{Obj}\, t.\langle\!\langle rt \mid \mathtt{x}{:}int, \mathtt{move}{:}int{\to}t\rangle\!\rangle$, and arises

from using the (second order) row variable r: instances of the `Obj`-type are obtained by substituting any *well-kinded* row (i.e. one that does not contain the names `x` and `move`) for the row rt that results from the application of r to the type t. In our system, instead, we place polymorphism outside rows: the type of `move` is polymorphic in the type-variable u, and instances of this type are obtained by substituting any *syntactically well-formed* type that matches the bound $\texttt{Obj}\, t.\langle\!\langle \texttt{x}{:}int, \texttt{move}{:}int{\to}t \rangle\!\rangle$ for u.

As it turns out, the correspondence between polymorphic `Obj`-types of the original system, and the type-variables of our systems carries over to typing derivations. As a matter of fact, the correspondence works correctly as long as the polymorphic `Obj`-types are used in a "disciplined", or *regular* way in typing judgements and derivations from [16]; it breaks, instead, in other cases. Consider, for instance, the following judgement:

$$r : T{\to}[m],\ e : \texttt{Obj}\, t.\langle\!\langle rt \rangle\!\rangle \vdash \langle e \leftarrow\!\!+\ m{=}\lambda s.s \rangle : \texttt{Obj}\, t.\langle\!\langle rt \mid m{:}t \rangle\!\rangle$$

While this judgement is derivable in [16], replacing the polymorphic `Obj`-types with the corresponding type variables fails to produce a derivable judgement in our system. There are several reasons for this, the most evident being that our (*ext*) rule requires a `Obj`-type (not a variable) in the type of the extended object.

In general, the correspondence between row and type variables breaks whenever the same row-variable occurs in different polymorphic `Obj`-types of a derivation. Fortunately, however, it can be shown that such undesired uses of polymorphic `Obj`-type may always be dispensed with in derivations for closed-object typing judgements of the form $\varepsilon \vdash e : \tau$. The proof of this fact is rather complex, and requires a number of technical lemmas establishing some useful properties of the typing derivations of [16]. Due to the lack of space, below we only state the main result, referring the reader to [4] for full details.

Definition 3. [4] Let $\varepsilon \vdash e : \tau$ be a judgement from [16], and let Ξ be a derivation for this judgement. We say that Ξ is *regular* if and only if every every row-variable of Ξ occurs always within the same polymorphic `Obj`-type in Ξ.

Theorem 4. [4] *Every judgement of the form $\varepsilon \vdash e : \tau$ has a derivation in [16] iff it has a regular derivation.*

Using this result, we may then prove that every typing judgement of the form $\varepsilon \vdash e : \tau$ derivable in [16] is derivable also in our system. We do this, in the next subsection, introducing an encoding function that translates every regular derivation from [16] into a corresponding derivation in our system. Restricting to regular derivation is convenient in that it greatly simplifies the definition of the encoding; on the other hand, given the result established in Theorem 4, it involves no loss of generality for the judgements of interest.

4.1 Encoding the *Fisher-Honsell-Mitchell* Type System

Throughout this subsection, types, contexts, judgements and derivations from [16] will be referred to as, respectively row–types, row-contexts, row-judgements. Also we will implicitly assume that they occur within regular derivations.

Encoding of Types, Contexts and Judgements. The encoding of a row-type τ is the type τ^* that results from replacing every occurrence of a polymorphic `Obj`-type with a corresponding type-variable. The correspondence between the polymorphic `Obj`-types of τ and the type-variables of τ^* may be established using any injective map from row-variables to corresponding type variables (this is because in every regular row-derivation there is a one-to-one correspondence between row-variables and polymorphic `Obj`-types). To ease the presentation, we will thus assume that the row-variables of regular derivations are all chosen from a given set $\mathcal{V}_r$, and that an injective map $\xi : \mathcal{V}_r \mapsto \mathcal{V}_t$ is given, where $\mathcal{V}_t$ is a corresponding set of type variables.

Definition 5 [Encoding of Types]**.** Let τ be a row-type. The encoding of τ, denoted by τ^*, is defined as follows:

- $\tau^* = \tau$, for every type variable or type constant τ;
- $(\tau_1 \to \tau_2)^* = \tau_1^* \to \tau_2^*$
- $(\mathtt{Obj}\, t.\langle\!\langle m_1{:}\tau_1, \ldots, m_k{:}\tau_k \rangle\!\rangle)^* = \mathtt{Obj}\, t.\langle\!\langle m_1{:}\tau_1^*, \ldots, m_k{:}\tau_k^* \rangle\!\rangle$
- $(\mathtt{Obj}\, t.\langle\!\langle rt \mid m_1{:}\tau_1, \ldots, m_k{:}\tau_k \rangle\!\rangle)^* = \xi(r)$

It follows from the definition that the encoding of a "ground" type, i.e. one that does not contain any occurrence of row-variables, leaves the type unchanged: in other words, τ^* is equal to τ, whenever τ is ground in the sense we just explained.

The encoding of a row-context Γ is more elaborate, and it is given with respect to the regular derivation where the context occurs.

Definition 6 [Encoding of Contexts]**.** Let Γ be a row-context of a regular derivation Ξ, and let $\mathtt{Obj}\, t.\langle\!\langle rt \mid \overline{m{:}\tau} \rangle\!\rangle$ denote *the* polymorphic `Obj`-type of Ξ where r occurs. The encoding of Γ, denoted by Γ^*_Ξ, is defined as follows:

- $\varepsilon^*_\Xi = \varepsilon$
- $(\Gamma, t : T)^*_\Xi = \Gamma^*_\Xi$;
- $(\Gamma, r : \kappa)^*_\Xi = \Gamma^*_\Xi,\ \xi(r) \lessdot\!\!\# \mathtt{Obj}\, t.\langle\!\langle \overline{m{:}\tau}^* \rangle\!\rangle$
- $(\Gamma, x : \tau)^*_\Xi = \Gamma^*_\Xi,\ x : \tau^*$.

The definition is well-posed as every row-variable occurs in just one polymorphic `Obj`-type, Ξ being regular. Also note that the encoding of every valid row-context Γ is a valid context, since every row-variable r may occur at most once in Γ.

The encoding of row-judgements is also given with respect to the regular derivation where they occur.

Definition 7 [Encoding of Judgements]**.** Let $\Gamma \vdash e : \tau$ be a row-judgement of a regular derivation Ξ. Then the encoding of this row-judgement is $\Gamma^*_\Xi \vdash e : \tau^*$.

Note that if $\varepsilon \vdash e : \tau$ is a derivable judgement, then its encoding is the judgement $\varepsilon \vdash e : \tau$ itself. This is easily seen as the type τ must be ground, the judgement being derivable.

Encoding of Type Rules. The row-portion of the type system from [16] has no counterpart in our system, as we axiomatize well-formedness for rows syntactically. The encoding of a type rule for terms, instead, is the result of encoding the row-judgements in the conclusion and in the premises of the rule, with the exception of the rules (*send*), (*ext*) and (*over*).

Definition 8 [ENCODING OF (*send*)]**.**

$$\left(\frac{\Gamma \vdash e : \mathtt{Obj}\, t.\langle\!\langle R \mid \overline{m{:}\tau}\rangle\!\rangle}{\Gamma \vdash e \Leftarrow m : [\mathtt{Obj}\, t.\langle\!\langle R \mid \overline{m{:}\tau}\rangle\!\rangle/t]\tau}\right)^*_{\Xi} = \frac{\Gamma^*_\Xi \vdash e : (\mathtt{Obj}\, t.\langle\!\langle R \mid \overline{m{:}\tau}\rangle\!\rangle)^* \quad \Gamma^*_\Xi \vdash (\mathtt{Obj}\, t.\langle\!\langle R \mid \overline{m{:}\tau}\rangle\!\rangle)^* \lessdot\!\!\# \mathtt{Obj}\, t.\langle\!\langle m{:}\tau^*\rangle\!\rangle}{\Gamma^*_\Xi \vdash e \Leftarrow m : ([\mathtt{Obj}\, t.\langle\!\langle R \mid \overline{m{:}\tau}\rangle\!\rangle/t]\tau)^*}$$

where Ξ is the regular derivation where the rule occurs.

The type $(\mathtt{Obj}\, t.\langle\!\langle R \mid \overline{m{:}\tau}\rangle\!\rangle)^*$ may either be a a $\mathtt{Obj}$-type, if R is a list of $m{:}\tau$ pairs, or the type variable $u = \xi(r)$ if $R = \langle\!\langle rt \mid \ldots\rangle\!\rangle$. The two cases correspond respectively to messages to an object, and messages to *self*. As in [16], they are treated uniformly in our type system.

Definition 9 [ENCODING OF (*ext*)]**.**

$$\left(\frac{\begin{array}{l}\Gamma \vdash e_1 : \mathtt{Obj}\, t.\langle\!\langle R \mid \overline{m{:}\tau}\rangle\!\rangle \\ \Gamma,\, r : T \to [\overline{m}, n] \vdash \qquad r \textit{ not in } \tau \\ \qquad e_2 : [\mathtt{Obj}\, t.\langle\!\langle rt \mid \overline{m{:}\tau}, n{:}\tau\rangle\!\rangle/t](t \to \tau)\end{array}}{\Gamma \vdash \langle e_1 \longleftarrow n = e_2\rangle : \mathtt{Obj}\, t.\langle\!\langle R \mid \overline{m{:}\tau}, n{:}\tau\rangle\!\rangle}\right)^*_{\Xi} =$$

$$= \frac{\Gamma^*_\Xi \vdash e_1 : \mathtt{Obj}\, t.\langle\!\langle R^* \mid \overline{m{:}\tau^*}\rangle\!\rangle \quad n \notin \mathcal{M}(R*) \cup \{\overline{m}\} \quad \Gamma^*_\Xi,\, u \lessdot\!\!\# \mathtt{Obj}\, t.\langle\!\langle \overline{m{:}\tau^*}, n{:}\tau^*\rangle\!\rangle \vdash e_2 : [u/t](t \to \tau)^*}{\Gamma^*_\Xi \vdash \langle e_1 \longleftarrow n = e_2\rangle : \mathtt{Obj}\, t.\langle\!\langle R^* \mid \overline{m{:}\tau^*}, n{:}\tau^*\rangle\!\rangle}$$

where $u = \xi(r)$, and Ξ is the regular derivation where the rule occurs.

Note that the context $\Gamma^*_\Xi,\, u \lessdot\!\!\# \mathtt{Obj}\, t.\langle\!\langle \overline{m{:}\tau^*}\rangle\!\rangle$ is just the encoding, in Ξ, of the row-context $\Gamma,\, r : T \to [\overline{m}]$. The notation R^* is consistent because, as we show in [4], for every instance of the (*ext*) rule occurring in a regular derivation it must be the case that R is a ground row of the form $\overline{p{:}\sigma}$ for some methods $\overline{p}$ and types $\overline{\sigma}$. The encoding of (*over*) is defined similarly to the (*ext*) case.

Theorem 10 [COMPLETENESS]**.** *Let $\varepsilon \vdash e : \tau$ be a row-judgement derivable in [16]. Then $\varepsilon \vdash e : \tau$ is derivable in our system.*

Proof. Since $\varepsilon \vdash e : \tau$ is derivable, the encoding of this judgement coincides with the judgement itself. Let then Ξ be a regular derivation for $\varepsilon \vdash e : \tau$ (the existence of such a derivation follows from Theorem 4): to prove the claim, it is enough to show that the encoding (in Ξ) of every other judgement of Ξ is derivable in our system.

Let then $\Gamma' \vdash e' : \tau'$ be a judgement of Ξ, and let Ξ' be the sub-derivation of Ξ rooted at $\Gamma' \vdash e' : \tau'$. The proof is by induction on Ξ'. The basis of induction, the (*projection*) case, follows immediately, and most of the inductive cases follow easily by induction.

The only slightly more elaborate cases is when Ξ' ends up with (*send*). In this case the last rule of Ξ', ans its encoding are, respectively:

$$\left(\frac{\Gamma \vdash e : \mathtt{Obj}\, t.\langle\!\langle R \mid \overline{m{:}\tau} \rangle\!\rangle}{\Gamma \vdash e \Leftarrow m : [\mathtt{Obj}\, t.\langle\!\langle R \mid \overline{m{:}\tau} \rangle\!\rangle / t]\tau}\right)^*_{\Xi} \text{ and } \frac{\Gamma^*_\Xi \vdash e : (\mathtt{Obj}\, t.\langle\!\langle R \mid \overline{m{:}\tau} \rangle\!\rangle)^* \quad \Gamma^*_\Xi \vdash (\mathtt{Obj}\, t.\langle\!\langle R \mid \overline{m{:}\tau} \rangle\!\rangle)^* \lessdot\!\!\# \ \mathtt{Obj}\, t.\langle\!\langle m{:}\tau^* \rangle\!\rangle}{\Gamma^*_\Xi \vdash e \Leftarrow m : ([\mathtt{Obj}\, t.\langle\!\langle R \mid \overline{m{:}\tau} \rangle\!\rangle / t]\tau)^*}$$

That $\Gamma^*_\Xi \vdash e : (\mathtt{Obj}\, t.\langle\!\langle R \mid \overline{m{:}\tau} \rangle\!\rangle)^*$ is derivable follows from the induction hypothesis. For the other judgement in the premise, instead, we distinguish the two possible sub-cases. If $(\mathtt{Obj}\, t.\langle\!\langle R \mid \overline{m{:}\tau} \rangle\!\rangle)^* = \mathtt{Obj}\, t.\langle\!\langle R^* \mid \overline{m{:}\tau^*} \rangle\!\rangle$ then the judgement is question is derivable directly by ($\lessdot\!\!\#$). If, instead, $(\mathtt{Obj}\, t.\langle\!\langle R \mid \overline{m{:}\tau} \rangle\!\rangle)^*$ is a type variable, say $\xi(r)$, then it must be the case that $R = \langle\!\langle rt \mid \overline{p{:}\sigma} \rangle\!\rangle$ for some $\overline{p{:}\sigma}$. But then $r \in Dom(\Gamma)$ and $\mathtt{Obj}\, t.\langle\!\langle rt \mid \overline{p{:}\sigma}, \overline{m{:}\tau} \rangle\!\rangle$ is the polymorphic `Obj`-type for r in Ξ. Hence, from Definition 6, we have that $\xi(r) \lessdot\!\!\# \ \mathtt{Obj}\, t.\langle\!\langle \overline{p{:}\sigma}, \overline{m{:}\tau} \rangle\!\rangle \in \Gamma^*_\Xi$, and then $\Gamma^*_\Xi \vdash (\mathtt{Obj}\, t.\langle\!\langle R \mid \overline{m{:}\tau} \rangle\!\rangle)^* \lessdot\!\!\# \ \mathtt{Obj}\, t.\langle\!\langle m{:}\tau^* \rangle\!\rangle$ is derivable by ($\lessdot\!\!\#$ *proj*) and ($\lessdot\!\!\#$ *trans*). □

5 Classes

Introducing classes relies on the idea of distinguishing two kinds of types, `Class` types whose elements are classes, and `Obj` types whose elements are instances created by the classes. This distinction is inspired by [17], but we use it here in a completely different way and with orthogonal purposes.

5.1 Class Types and Object Types

Object types have the usual form $\mathtt{Obj}\, t.\langle\!\langle m^\iota_1{:}\tau_1, \ldots, m^\iota_k{:}\tau_k \rangle\!\rangle$, with the difference that we now require that the component methods be *instance-methods* annotated by the superscript $^\iota$. Class types, instead, have the form:

$$\mathtt{Class}\, t.\langle\!\langle m^c_1{:}\tau_1, \ldots, m^c_k{:}\tau_k, m^\iota_1{:}\sigma_1, \ldots, m^\iota_l{:}\sigma_l \rangle\!\rangle$$

with the superscript ^{c} distinguishing *class-methods* from the remaining instance-methods. The intention is to have each class define a set of class-methods for exclusive use of the class and its sub-classes, as well as a set of instance-methods for the instances of the class (and of the class' sub-classes). Instance-methods may only invoke or override other instance-methods, so that classes are protected from updates caused by their instances; class-methods, instead, are not subject to this restriction.

Every class defines (or inherits from its super-classes) at least one class-method – *new* – for creating new instances of that class. As in [17], instances of a class are created by *packaging* a class, an operation that does nothing but

"sealing" the class, so that no methods may be added. Sealing a class changes its type into a corresponding `Obj` type, that results from hiding all of the class-methods of the class. As for subclassing, new classes may be derived from a class by adding new methods or redefining existing methods of that class.

To account for the above features, the object-related forms of the untyped calculus are extended as follows:

$$e ::= \mid \mathit{topclass} \mid \langle e_1 \overset{o}{\longleftarrow\!+} m{=}e_2 \rangle \mid \mathit{pack}(e) \mid \langle e_1 \leftarrow m{=}e_2 \rangle \mid e \Leftarrow m \mid e \hookleftarrow m$$

The operational meaning of the operations of override, send, and search is exactly as in Section 2. The meaning of the remaining expressions is given next.

topclass is a pre-defined constant representing the empty class and defined as follows: $\mathit{topclass} \equiv \langle \mathit{new}^c{=}\lambda s.\mathit{pack}(s) \rangle$. This class defines class-method *new* for creating instances: the result of a call to *new* on a class is the instance of that class that results from packaging the class.

New classes may be derived from the empty class by a sequence of method overrides and extensions. The symbol $\overset{o}{\longleftarrow\!+}$ above denotes two different operators, $\overset{c}{\longleftarrow\!+}$ and $\overset{\iota}{\longleftarrow\!+}$, denoting class-extension with, respectively, class-methods, and instance methods.

The operational semantics of the *pack* operator is defined by few additional cases of the reduction relation; the effect of packaging on types, in turn, is rendered in terms of a corresponding type operator, denoted by `pack`. The new cases of reduction for *pack* and the equational rules for `pack`aged types are defined below:

$$\begin{array}{lll}
\mathit{pack}(c) & \overset{eval}{\longrightarrow} c & \texttt{pack}(b) = b \\
\mathit{pack}(\lambda x.e) & \overset{eval}{\longrightarrow} \lambda x.e & \texttt{pack}(\tau_1 \to \tau_2) = \tau_1 \to \tau_2 \\
\mathit{pack}(e) \Leftarrow m & \overset{eval}{\longrightarrow} (e \hookleftarrow m)\,\mathit{pack}(e) & \texttt{pack}(\texttt{Class}\,t.\langle \overline{m^c{:}\rho}, \overline{p^\iota{:}\tau} \rangle) = \texttt{Obj}\,t.\langle \overline{p^\iota{:}\tau} \rangle \\
\mathit{pack}(\mathit{pack}(e)) & \overset{eval}{\longrightarrow} \mathit{pack}(e) & \texttt{pack}(\texttt{Obj}\,t.R) = \texttt{Obj}\,t.R \\
 & & \texttt{pack}(\texttt{pack}(\tau)) = \texttt{pack}(\tau)
\end{array}$$

Packaging a class produces the corresponding `Obj` type that results from hiding all of the class-methods of the class. Packaging any other expression, instead, has no effect on the type of the expression. The introduction of the `pack` operator induces a new, and richer, notion of type equality that now allows two types to be identified if they are equal modulo the equational rules for the `pack` operator.

The definition of matching is readily extended to the newly introduced types. Matching over `Class`-types is exactly as matching over `Obj`-types, namely:

$$\frac{\Gamma \vdash * \qquad \langle \overline{m^o{:}\tau}, \overline{n^o{:}\sigma} \rangle \ \mathit{well\ formed}}{\Gamma \vdash \texttt{Class}\,t.\langle \overline{m^o{:}\tau}, \overline{n^o{:}\sigma} \rangle \lessdot\!\# \ \texttt{Class}\,t.\langle \overline{m^o{:}\tau} \rangle} \ (\lessdot\!\#\ \mathit{class}),$$

where ^{o} may either be ^{c} or $^\iota$. As for matching between `Class` and `Obj` types, we have the following rule:

$$\frac{\Gamma \vdash * \qquad \langle \overline{m^c{:}\tau}, \overline{n^\iota{:}\sigma} \rangle \ \mathit{well\ formed}}{\Gamma \vdash \texttt{Class}\,t.\langle \overline{m^c{:}\tau}, \overline{n^\iota{:}\sigma} \rangle \lessdot\!\# \ \texttt{Obj}\,t.\langle \overline{n^\iota{:}\sigma} \rangle} \ (\lessdot\!\#\ \mathit{obj}),$$

that is, every `Class` type matches its corresponding `Obj` type.

5.2 Typing Rules

The new types require few additional changes in the object-related portion of the type system, which are described next. The following two rules define the type of the empty class and of packaged expressions.

$$\frac{\Gamma \vdash *}{\Gamma \vdash \mathit{topclass} : \mathtt{Class}\, t.\langle\!\langle \mathit{new}^c : \mathtt{pack}(t) \rangle\!\rangle} \quad (\mathit{topclass}).$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathit{pack}(e) : \mathtt{pack}(\tau)} \quad (\mathit{pack}).$$

For the remaining object-expression, we need different rules for distinguishing the different behavior of class-methods and instance-methods.

Class Methods. The typing of class-method invocation is as follows:

$$\frac{\Gamma \vdash e : \sigma \quad \Gamma \vdash \sigma <\!\!\# \ \mathtt{Class}\, t.\langle\!\langle n^c : \tau \rangle\!\rangle}{\Gamma \vdash e \Leftarrow n : [\sigma/t]\tau} \quad (\mathit{c\text{-}send}),$$

If n is a class-method (as indicated by the annotation ^{c}), then e must have a `Class`-type matching $\mathtt{Class}\, t.\langle\!\langle n^c : \tau \rangle\!\rangle$. Therefore, only class-methods may be invoked on a class; class-methods may, instead, perform (internal) invocations or overrides also on instance-methods of that class. The following rule for class-method addition allows this behaviour.

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \mathtt{Class}\, t.\langle\!\langle R \mid \overline{m^o : \rho} \rangle\!\rangle \quad n \notin \mathcal{M}(R) \cup \{\overline{m}\} \\ \Gamma,\ u <\!\!\# \ \mathtt{Class}\, t.\langle\!\langle \overline{m^o : \rho}, n^c : \tau \rangle\!\rangle \vdash e_2 : [u/t](t \to \tau)\end{array}}{\Gamma \vdash \langle e_1 \overset{c}{\leftarrow\!\!+} n = e_2 \rangle : \mathtt{Class}\, t.\langle\!\langle R \mid \overline{m^o : \rho}, n^c : \tau \rangle\!\rangle} \quad (\mathit{c\text{-}ext}).$$

As usual, method addition, is subject to the constraint that the n method be not already in the type of class. The typing rule for method override is similar to (*c-ext*), while the typing of search expressions follows the same idea as (*c-send*) above.

Instance Methods. As we said, instance-method may only invoke other instance-methods of the *self* object. The following typing rule enforces this constraint:

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \mathtt{Class}\, t.\langle\!\langle R \mid \overline{m^\iota : \rho} \rangle\!\rangle \qquad n \notin \mathcal{M}(R) \cup \{\overline{m}\} \\ \Gamma,\ u <\!\!\# \ \mathtt{Obj}\, t.\langle\!\langle \overline{m^\iota : \rho}, n^\iota : \tau \rangle\!\rangle \vdash e_2 : [\mathtt{pack}(u)/t](t \to \tau)\end{array}}{\Gamma \vdash \langle e_1 \overset{\iota}{\leftarrow\!\!+} n = e_2 \rangle : \mathtt{Class}\, t.\langle\!\langle R \mid \overline{m^\iota : \rho}, n^\iota : \tau \rangle\!\rangle} \quad (\iota\text{-}\mathit{ext}).$$

Note that the bound for u is so defined as to allow the n method to be applied on objects of every type matching the `Obj` type corresponding to the `Class` type (as in ($<\!\!\#$ *obj*) rule). This way, instance-methods may safely be inherited by

the instances of the class. Note, furthermore, that the polymorphic type of e_2 is so defined as to ensure that e_2 may only be applied on elements of packaged types (i.e., objects).

A corresponding constraint is imposed on the typing of instance-method invocation.

$$\frac{\Gamma \vdash e : \mathtt{pack}(\sigma) \quad \Gamma \vdash \sigma \lessdot\!\!\# \ \mathtt{Obj}\, t.\langle\!\langle n^{\iota}{:}\tau \rangle\!\rangle}{\Gamma \vdash e \Leftarrow n : [\mathtt{pack}(\sigma)/t]\tau} \quad (\iota\text{-}send),$$

The type of the receiver must be a packaged type consistent with the polymorphic type that is associated with the body of the method. The rules for method search and override may be defined following this idea.

5.3 A Simple Example

Using the above type rules, it is now possible to define an object expression representing the class of points with an x coordinate and, say, a *set* method for updating the position of the points in the class. Assuming that *create* is a class method, and that x and *set* are instance methods, the class of points may be defined as follows:

$$\begin{array}{rcll} \mathtt{ptClass} & \equiv & \langle\, new & = \lambda s.pack(s); \\ & & create & = \lambda s.\lambda v.((s \Leftarrow new) \Leftarrow set\, v); \\ & & x & = \lambda s.0; \\ & & set & = \lambda s.\lambda v.\langle s \leftarrow x{=}\lambda self\,.v \rangle\, \rangle \end{array}$$

where the value of the x field is defaulted to 0 in the class definition. The following type may then be derived:

$$\mathtt{ptClass} : \mathtt{Class}\, t.\langle\!\langle new^{c}{:}\mathtt{pack}(t),\ create^{c}{:}\mathtt{pack}(t),\ x^{\iota}{:}int,\ set^{\iota}{:}int{\to}\mathtt{pack}(t) \rangle\!\rangle$$

Instances of this class may then be created with a *create* message to `ptclass`. For instance, the expression ($\mathtt{ptclass} \Leftarrow create\, 3$) creates a point instance of type $\mathtt{Obj}\, t.\langle\!\langle x^{\iota}{:}int,\ set^{\iota}{:}int{\to}\mathtt{pack}(t) \rangle\!\rangle$, with x coordinate valued 3 and a *set* method.

6 Conclusions

We have presented a new type system for the *Lambda Calculus of Objects*. The main difference with respect to the original proposal, is that in [16] method polymorphism is rendered in terms of quantification over row-schemas, whereas in our system it is captured by means of match-bounded quantification and matching. A formal analysis of the relative expressive power of the two systems shows that they coincide on derivations for closed-object typing judgements. On the other hand, there are some fundamental tradeoffs between the two approaches, both in terms of complexity and of their logical rendering. In fact, while the new solution appears to reduce the complexity of the system, freeing it from the calculus of rows of the original system, on the other hand the auxiliary judgements

and side-conditions needed for matching are not costless, and may have undesired consequences in the encoding of the new system in Logical Frameworks [15].

We have then presented an extension of the new system that gives provision for classes in an object-based setting. The extended calculus may, in some respects, be seen as a functional counterpart of the *Imperative Object Calculus* of [1]. Besides the different operational setting (i.e. imperative versus functional), that proposal differs from ours in that, while using matching to model method inheritance between classes, it relies on subtyping for the treatment of self types. Instead, our system requires no subtyping, since it relies on matching as the only relation over class and object types.

Acknowledgements. We would like to thank Mariangiola Dezani-Ciancaglini for inspiring this work and for endless discussions on earlier drafts. Suggestions from the anonymous referees helped to improve the presentation substantially.

References

1. M. Abadi and L. Cardelli. An Imperative Objects Calculus. In P.D. Mosses, M. Nielsen, and M.I. Schwartzbach, editors, *Proceedings of TAPSOFT'95: Theory and Practice of Software Development*, volume 915 of *LNCS*, pages 471–485. Springer–Verlag, May 1995.
2. M. Abadi and L. Cardelli. On Subtyping and Matching. In *Proceedings of ECOOP'95: European Conference on Object-Oriented Programming*, volume 952 of *LNCS*, pages 145–167. Springer–Verlag, August 1995.
3. M. Abadi and L. Cardelli. A Theory of Primitive Objects. *Information and Computation*, 125(2):78–102, March 1996.
4. V. Bono and M. Bugliesi. Matching Lambda Calculus of Objects. Submitted for publication, 1996.
5. V. Bono, M. Bugliesi, and L. Liquori. A Lambda Calculus of Incomplete Objects. In *Proc. of MFCS*, volume 1113 of *Lecture Notes in Computer Science*, pages 218–229. Springer-Verlag, 1996.
6. V. Bono, M. Bugliesi, M.Dezani, and L. Liquori. Subtyping Constraints for Incomplete Objects. In *Proc. of CAAP*, Lecture Notes in Computer Science. Springer-Verlag, 1997. To appear.
7. 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.
8. K.B. Bruce. A Paradigmatic Object-Oriented Programming Language: Design, Static Typing and Semantcs. *Journal of Functional Programming*, 1(4):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. A Self-ish Model of Inheritance. Manuscript, 1987.
13. W. Cook, W. Hill, and P. Canning. Inheritance is not Subtyping. In *Proc. of ACM Symp. POPL*, pages 125–135. ACM Press, 1990.
14. W.R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.
15. Harper R. Honsell F. and Plotkin G. A Framework for Defining Logics. *J.ACM*, 40(1):143–184, 1993.
16. 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.
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. M. Wand. Complete Type Inference for Simple Objects. In *Proc. of IEEE Symp. LICS*, pages 37–44. Silver Spring, 1987.

A Operational Semantics

$$
\begin{array}{lll}
(\beta) & (\lambda x.e_1)\, e_2 & \stackrel{eval}{\longrightarrow} \; [e_2/x]\, e_1 \\
(\Leftarrow) & e \Leftarrow m & \stackrel{eval}{\longrightarrow} \; (e \hookleftarrow m)\, e \\
(\hookleftarrow \mathit{succ}) & \langle e_1 \hookleftarrow\!\circ\ m{=}e_2 \rangle \hookleftarrow m & \stackrel{eval}{\longrightarrow} \; e_2 \\
(\hookleftarrow \mathit{next}) & \langle e_1 \hookleftarrow\!\circ\ n{=}e_2 \rangle \hookleftarrow m & \stackrel{eval}{\longrightarrow} \; e_1 \hookleftarrow m \\
 & & \\
(\mathit{fail}\ \langle\rangle) & \langle\rangle \hookleftarrow m & \stackrel{eval}{\longrightarrow} \; \mathit{err} \\
(\mathit{fail}\ \mathit{abs}) & \lambda x.e \hookleftarrow m & \stackrel{eval}{\longrightarrow} \; \mathit{err} \\
(\mathit{err} \hookleftarrow\!\circ\ ) & \langle \mathit{err} \hookleftarrow\!\circ\ m = e \rangle & \stackrel{eval}{\longrightarrow} \; \mathit{err} \\
(\mathit{err}\ \mathit{abs}) & \lambda x.\mathit{err} & \stackrel{eval}{\longrightarrow} \; \mathit{err} \\
(\mathit{err}\ \mathit{app}) & \mathit{err}\ e & \stackrel{eval}{\longrightarrow} \; \mathit{err} \\
(\mathit{err} \hookleftarrow) & \mathit{err} \hookleftarrow n & \stackrel{eval}{\longrightarrow} \; \mathit{err}
\end{array}
$$

B Typing Rules

General Rules: $\Gamma \vdash A$ stands for any derivable judgement in the system.

(*start*)
$$\frac{}{\varepsilon \vdash *}$$

(*projection*)
$$\frac{\Gamma \vdash * \quad x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

(*weakening*)
$$\frac{\Gamma \vdash A \quad \Gamma, \Gamma' \vdash *}{\Gamma, \Gamma' \vdash A}$$

Rules for Terms

$$(var)\quad \frac{\Gamma \vdash *\quad x \notin Dom(\Gamma)}{\Gamma,\ x{:}\tau \vdash *}$$

$$(abs)\quad \frac{\Gamma,\ x{:}\tau_1 \vdash e{:}\tau_2}{\Gamma \vdash \lambda x.e{:}\tau_1{\to}\tau_2}$$

$$(app)\quad \frac{\Gamma \vdash e_1{:}\tau_1{\to}\tau_2 \quad \Gamma \vdash e_2{:}\tau_1}{\Gamma \vdash e_1 e_2{:}\tau_2}$$

$$(empty)\quad \frac{\Gamma \vdash *}{\Gamma \vdash \langle\rangle : \mathtt{Obj}\, t.\langle\!|\rangle}$$

$$(ext)\quad \frac{\begin{array}{l}\Gamma \vdash e_1 : \mathtt{Obj}\, t.\langle\!| R \mid \overline{m{:}\rho}|\!\rangle \\ \Gamma,\ u \lessdot\!\!\# \ \mathtt{Obj}\, t.\langle\!|\overline{m{:}\rho}, n{:}\tau|\!\rangle \vdash e_2 : [u/t](t{\to}\tau) \qquad n \notin \mathcal{M}(\langle\!| R \mid \overline{m{:}\tau}|\!\rangle)\end{array}}{\Gamma \vdash \langle e_1 {\leftarrow\!\!+}\ n{=}e_2\rangle : \mathtt{Obj}\, t.\langle\!| R \mid \overline{m{:}\rho}, n{:}\tau|\!\rangle}$$

$$(over)\quad \frac{\begin{array}{l}\Gamma \vdash e_1 : \sigma \qquad \Gamma \vdash \sigma \lessdot\!\!\# \ \mathtt{Obj}\, t.\langle\!|\overline{m{:}\rho}, n{:}\tau|\!\rangle \\ \Gamma, u \lessdot\!\!\# \ \mathtt{Obj}\, t.\langle\!|\overline{m{:}\rho}, n{:}\tau|\!\rangle \vdash e_2 : [u/t](t{\to}\tau)\end{array}}{\Gamma \vdash \langle e_1 {\leftarrow}\ n{=}e_2\rangle : \sigma}$$

$$(send)\quad \frac{\Gamma \vdash e : \sigma \quad \Gamma \vdash \sigma \lessdot\!\!\# \ \mathtt{Obj}\, t.\langle\!| n{:}\tau|\!\rangle}{\Gamma \vdash e \Leftarrow n : [\sigma/t]\tau}$$

$$(search)\quad \frac{\Gamma \vdash e : \sigma \quad \Gamma \vdash \sigma \lessdot\!\!\# \ \mathtt{Obj}\, t.\langle n{:}\tau\rangle \quad \Gamma \vdash \varsigma \lessdot\!\!\# \ \sigma}{\Gamma \vdash e \hookleftarrow n : [\varsigma/t](t{\to}\tau)}$$

Rules for Matching

$$(\lessdot\!\!\# \ \ var)\quad \frac{\Gamma \vdash *\quad u \notin \Gamma \quad u \notin \tau}{\Gamma,\ u \lessdot\!\!\# \ \tau \vdash *}$$

$$(\lessdot\!\!\# \ \ proj)\quad \frac{\Gamma \vdash *\quad u \lessdot\!\!\# \ \tau \in \Gamma}{\Gamma \vdash u \lessdot\!\!\# \ \tau}$$

$$(\lessdot\!\!\# \ \ refl)\quad \frac{\Gamma \vdash *}{\Gamma \vdash \tau \lessdot\!\!\# \ \tau}$$

$$(\lessdot\!\!\# \ \ trans)\quad \frac{\Gamma \vdash \sigma \lessdot\!\!\# \ \tau \quad \Gamma \vdash \tau \lessdot\!\!\# \ \rho}{\Gamma \vdash \sigma \lessdot\!\!\# \ \rho}$$

$$(\lessdot\!\!\# \ )\quad \frac{\Gamma \vdash * \qquad \langle\!|\overline{m{:}\tau}, \overline{n{:}\sigma}|\!\rangle \ well\ formed}{\Gamma \vdash \mathtt{Obj}\, t.\langle\!|\overline{m{:}\tau}, \overline{n{:}\sigma}|\!\rangle \lessdot\!\!\# \ \mathtt{Obj}\, t.\langle\!|\overline{m{:}\tau}|\!\rangle}$$