A Typed Lambda Calculus of Objects

(Extended Abstract)

Luigi Liquori[1] and Giuseppe Castagna[2]

[1] Dipartimento d'Informatica, Università di Torino C.so Svizzera 185, 10149 Torino, Italy
[2] CNRS, LIENS-DMI, École Normale Supérieure, 45 rue d'Ulm, 75005 Paris, France

Abstract. In this paper, we present an explicitly typed version of the Lambda Calculus of Objects of [7], which is a development of the object-calculi defined in [10, 2]. This calculus supports *object extension* in presence of *object subsumption*. Extension is the ability of modify the behavior of an object by adding new methods (and inheriting the existing ones). Object subsumption allows to use objects with a bigger interface in a context expecting another object with a smaller interface. This calculus has a sound and decidable type system, "width" subtyping, and it allows for *first-class* method bodies.

1 Introduction

λ_o, the Lambda Calculus of Objects of [7], is an untyped[3] kernel calculus defined to give a foundation to object-based languages. This calculus essentially is an untyped λ-calculus enriched with object primitives. Objects are built up from an *empty object* by adding new methods or overriding existing ones. A primitive to call the methods of the objects is provided. The calculus supports a simple inheritance mechanism, a straightforward *mytype* method specialization, and dynamic lookup of methods. Its operational semantics is very simple and allows to treat the special symbol *self* of object-oriented languages directly by lambda abstraction. This calculus, however, lacks one of the most important features of object-oriented programming, namely the subtyping mechanism. Unfortunately, in λ_o it is not possible to use an object with some methods where an object with less methods is expected (this kind of relation is called "width" subtyping).

In [2], an extension, called $\lambda_o^{\prec}$, of this calculus was presented, by introducing a subtyping relation compatible with method extension. Subtyping is subject to the restriction that a method can be forgotten only if the remaining methods in the object do not refer to it. This is obtained by "labeling" the type of a method n by the names of the methods of the object that n uses.

However, we conjecture that the type systems of [7, 2] are undecidable. More precisely, recall that those type systems are defined for an untyped calculus. Therefore, in order to use those type systems in practice, it is necessary to have a type assignment algorithm that returns the principal type (supposing that a notion of principality can be defined) of every typable untyped term. Hitherto, no such an algorithm exists. Furthermore, by our experience, we believe that this algorithm cannot exist, the reason being the impossibility of assigning the right type to *self*, whose semantics, as clearly stated in [11], is context dependent.

For these reasons in this paper we define an explicitly typed version of $\lambda_o^{\prec}$, that we call $\lambda_o^{\prec t}$. We define a type system for $\lambda_o^{\prec t}$, we prove that it statically ensures type safety of

[3] By *untyped* we intend that the terms of the calculus are not annotated by types. This does not mean that there is no type system at all.

well-typed programs and we define a type-checking algorithm that is sound and complete with respect to the type system, and terminating (which implies the decidability of the type system). What we obtain, then, is the first known-to-be decidable type system for an object calculus that supports object extension. The introduction of explicit universal quantification (in contrast with the implicit universal quantification of [7, 2]) allows to write *first-order* method bodies that can be passed as function arguments. This increase the expressiveness of the language, since it allows to write "portable methods".

Then we study the relation between $\lambda_o^{\prec}$ and $\lambda_o^{\prec t}$: we show that $\lambda_o^{\prec t}$ is at least as expressive as $\lambda_o^{\prec}$ (since there exists a *type erasing function* from typed to untyped terms, such that every untyped expression typeable in $\lambda_o^{\prec}$ is the erasure of some typed expression typeable in $\lambda_o^{\prec t}$) and that $\lambda_o^{\prec}$ can be used as a target language to compile $\lambda_o^{\prec t}$ (since the computation of the type-erasure of a term e of $\lambda_o^{\prec t}$, returns the erasure of a reductum of e).

From a pragmatic point of view the situation can be described as follows. If the programmer writes an untyped program in $\lambda_o^{\prec}$, then we cannot define a type-checking algorithm that can statically ensure type safety of the program. Therefore, we ask the programmer to add some "type annotations" to his program. Thanks to these annotations it is possible to type check the program by using the algorithm we describe in this paper. Once that the safety of the program is statically ensured, type annotations can be erased and the execution can be computed on the untyped term: a theorem ensures that each step of the untyped computation is the erasure of the typed computation.

The work we present here has however some deeper motivations. The calculus described here stems from the research done by the authors on the addition of multi-methods to object-based languages. Multi-methods are methods whose code is selected according not only to the class of the receiver but also to the class of possible further arguments. In [3], one of the authors showed that multi-methods can be used to overcome the longstanding problem of specializing binary methods. According to [3] this can be obtained by adding special overloaded functions that use a late binding discipline. An overloaded function is a function that selects some code according to the type of its argument. Therefore, in order to execute an overloaded function one needs to compute the type of its argument. Since this was not possible in $\lambda_o^{\prec}$, we defined $\lambda_o^{\prec t}$, the calculus we describe in this paper. The extension of $\lambda_o^{\prec t}$ with multi-methods is described in a companion paper [6].

This paper is organized as follows. In Section 2, we present $\lambda_o^{\prec t}$, its operational semantics and type system. Some examples, showing the expressivity of this calculus, are provided in Section 3. Section 4 illustrates the main properties of the system, namely the subject reduction, the type-soundness theorems, and some theorems that relate $\lambda_o^{\prec t}$ with $\lambda_o^{\prec}$. In Section 5, an algorithmic presentation of our type system enjoying the minimum type property is described (although, formally, it should have been presented first). A section devoted to related work close this paper.

2 The Typed Calculus of Objects

In this section, we present the language $\lambda_o^{\prec t}$, its operational semantics and type system.

2.1 Types, Rows, Kinds and Expressions

The set of types, rows, kinds and expressions are mutually defined by the following grammar:

$$
\begin{array}{lllllll}
\text{Types} & \tau & ::= & \iota \mid t \mid \tau\to\tau \mid \forall r{:}\kappa.\tau \mid \texttt{class}\, t.R & \text{Labels} & \Delta & ::= \{\} \mid \Delta\cup\{n\} \\
\text{Rows} & R & ::= & r \mid \langle\!\langle\rangle\!\rangle \mid \langle\!\langle R \mid n{:}\tau_\Delta\rangle\!\rangle \mid \lambda t{:}T.R \mid R\tau & \text{Kinds} & \kappa & ::= T \mid [\overline{n}] \mid T\to\kappa
\end{array}
$$

$$
\begin{array}{llll}
\text{Expressions} \ e & ::= & x \mid c \mid \lambda x{:}\tau.e \mid e_1e_2 \mid \lambda r{:}\kappa.e \mid eR \mid & \text{(higher-order typed } \lambda\text{-calc.)} \\
 & & \langle\rangle \mid \langle e_1 \leftarrow\!\!\circ\ n{=}e_2\rangle \mid \langle e_1 \leftarrow n{=}e_2\rangle \mid e \Leftarrow n \mid & \text{(object expressions)} \\
 & & e \overset{R}{\hookleftarrow} n \mid err & \text{(auxiliary expressions).}
\end{array}
$$

For object expressions we have the following intuitive meaning: $\langle\rangle$ is the empty object; $\langle e_1 \leftarrow\!\!\circ\ n{=}e_2\rangle$ extends e_1 with a method n of body e_2; $\langle e_1 \leftarrow n{=}e_2\rangle$ replaces the body of method n in e_1 by e_2 (provided that n is present in e_1); $e \Leftarrow n$ sends message n to the object e; $e \overset{R}{\hookleftarrow} n$ searches the body of method n into the object e under the row-context R, and *err* is the error object. The last two (auxiliary) expressions are mainly used to define the operational semantics, and in practice, *must not* be avalaible to the programmer (because of encapsulation motivations). More precisely, the expression $e \overset{R}{\hookleftarrow} n$ searches the body of method n into an object e which can be a proper sub-object of (say) e', whose class-type contains also the methods described into the row-abstraction R. So, the $\hookleftarrow$ operator is a *destructive* operator which go "throught" the object until it finds the searched method, but which keeps track of the (type of) the methods it skips by using the annotation R (see the operational semantics): as such, the type information R is needed in order to keep safe the self-application semantics.

Let us describe the types. Expressions of the higher-order typed lambda calculus are typed by type-constants (denoted by ι), type-arrows and higher-order types, as usual. Objects are typed by rows of methods. However, the recursive nature of *self* construction makes their type to be recursive. Thus, the type of an object is $\texttt{class}\, t.R$, where R is the row of the methods that compose the object, while t represent the (recursive) type of *self* (i.e. *mytype*). Note, however, that rows are composed by *labeled-types*, that is types that are indexed by a set of method names. Indeed, for every method n we record in rows not only the type of n, but also the names of those methods used directly by n. Thus the label of a method n fingers those methods that cannot be forgotten if the method n is present. This may be better illustrated using the well-known example of objects points. Such objects have x, y, and *mv* methods. If points have integer coordinates, then the functionality of a *point* object may be represented in [7] by the following class-type:

$$\texttt{class}\, t.\langle\!\langle x{:}int, y{:}int, mv{:}int\to int\to t\rangle\!\rangle.$$

2.2 Untyped Terms

In λ_o, the Untyped Lambda Calculus of Objects, an example of such objects is:

$$
\begin{array}{lll}
point & \stackrel{\triangle}{=} & \langle x{=}\lambda self.0, y{=}\lambda self.0, mv{=}\lambda self.\lambda dx.\lambda dy. \\
 & & \qquad \langle\langle self \leftarrow x{=}\lambda s.(self \Leftarrow x) + dx\rangle \leftarrow y{=}\lambda s.(self \Leftarrow y) + dy\rangle\rangle,
\end{array}
$$

where $\langle n_1{=}e_1, \ldots, n_k{=}e_k\rangle$ is an abbreviation of $\langle\langle \ldots \langle\langle\rangle \leftarrow\!\!\circ\ n_1{=}e_1\rangle \ldots\rangle \leftarrow\!\!\circ\ n_k{=}e_k\rangle$.

A significant aspect of the type system is that if we perform a method addition of a method *col* to build a *colored_point* object from *point*, then the type $(int\to int\to t)$ of method *mv* does not change syntactically. Instead, the meaning of this type changes, since before the *col* addition the bound variable *t* referred to an object of the same type as *point*, whereas after the addition *t* refers to an object of the same type as *colored_point*.

A more complex form of method specialization occurs when a method is overridden. Suppose, for the sake of the example, we want to to build a *diag_point* from the previous *point*, where *x* and *y* coordinates are equal. First we override the *mv* method so that it modifies the *x* and *y* coordinates only when the displacements *dx* and *dy* are equal:

$$\begin{aligned} dp \;\triangleq\; &\langle point \leftarrow mv{=}\lambda self.\lambda dx.\lambda dy.\ \text{if } dx \neq dy \text{ then } self \text{ else} \\ &\qquad \langle\langle self \leftarrow x{=}\lambda s.(self \Leftarrow x) + dx\rangle \leftarrow y{=}\lambda s.(self \Leftarrow y) + dy\rangle\rangle. \end{aligned}$$

Then, we redefine the *y* method with a new body that performs a lookup on the *x* component:

$$diag\_point \triangleq \langle dp \leftarrow y = \lambda self.self \Leftarrow x\rangle.$$

Now, since the *mv* method is implemented by using *y*, it follows that when we send a *mv* message to *diag_point*, the inherited *mv* will invoke the more specialized *y* method.

So, method extension and method override are operations that can modify dependencies inside objects. In the previous examples, the dependencies of *point* and *diag_point* are "*mv uses x and y*", and "*mv uses x and y, and y uses x*", respectively. These dependencies can be codified by using labeled-types. For example, in our system, the object *point* has the following class-type:

$$\texttt{class}\ t.\langle\!\langle x{:}int, y{:}int, mv{:}(int{\rightarrow}int{\rightarrow}t)_{x,y}\rangle\!\rangle,$$

(where *int* stands for $int_{\{\}}$ and $\tau_{x,y}$ for $\tau_{\{x\}\cup\{y\}}$) while the object *diag_point* has the following class-type:

$$\texttt{class}\ t.\langle\!\langle x{:}int, y{:}int_x, mv{:}(int{\rightarrow}int{\rightarrow}t)_{x,y}\rangle\!\rangle.$$

We can forget, by subtyping, those methods that are not used by other methods in the object, i.e. a method is *forgettable* if and only it does not appear in the labels of the types of the remaining methods.

Kinds are used to "type" rows. In particular we said that extension of an object *e* by a method *n* is allowed only if a method *n* is not already present in (the type of) *e*. Thus, in order to type object extensions, negative information is needed, namely we must know that a row *does not* contain a given method. For this reason in [7] the kind $[\overline{n}]$ has been introduced. Intuitively, the elements of kind $[\overline{n}]$ are rows that do not include method names $\overline{n}$. More details on labeled-types can be found in [2].

2.3 Typed Terms

In the introduction, we said that we conjecture the undecidability of type inference in λ_o and that to make it decidable we require the programmer to annote the terms by some types, obtaining a term of $\lambda_o^{\prec t}$, the calculus defined right above. For instance, the object *point* at the very beginning of Section 2.2, can be type-annotated and written in $\lambda_o^{\prec t}$ as shown below, where $T_{[\overline{n}]}$ is an abbreviation for $T{\rightarrow}[\overline{n}]$.

$$\begin{aligned} point \;\triangleq\; \langle\ & x{=}\lambda r_1{:}T_{[x]}.\lambda self_1{:}\langle\!\langle r_1 t \mid x{:}int\rangle\!\rangle.0,\ \ y{=}\lambda r_2{:}T_{[y]}.\lambda self_2{:}\langle\!\langle r_2 t \mid y{:}int\rangle\!\rangle.0, \\ & mv{=}\lambda r_3{:}T_{[x,y,mv]}.\lambda self_3{:}\langle\!\langle r_3 t \mid x{:}int, y{:}int, mv{:}(int{\rightarrow}int{\rightarrow}t)_{x,y}\rangle\!\rangle.\lambda dx{:}int.\lambda dy{:}int. \\ &\qquad \langle\langle self_3 \leftarrow x{=}(\lambda r_4{:}T_{[x]}.\lambda self_4{:}\langle\!\langle r_4 t \mid x{:}int\rangle\!\rangle.(self_3 \Leftarrow x) + dx)\rangle \\ &\qquad\quad \leftarrow y{=}(\lambda r_5{:}T_{[y]}.\lambda self_5{:}\langle\!\langle r_5 t \mid y{:}int\rangle\!\rangle.(self_3 \Leftarrow y) + dy)\rangle\rangle. \end{aligned}$$

Note in particular that the type of *self* is composed by a fixed part, where the method being defined plus those necessary to type the body are specified, and a variable part (denoted by $r_i\, t$), that contains "all the remaining methods of the object". These remaining methods will be passed by the system to the (polymorphic) body at the moment of its selection. For example, when selecting the method x of *point*, the system will pass to the body the term $\lambda t{:}\, T.\langle\!\langle y{:}int, mv{:}(int{\rightarrow}int{\rightarrow}t)_{x,y}\rangle\!\rangle$. Note that this part must be left variable since it must take into account also the methods that are defined *after* the method at issue. This machinery is handled by a suitable typed operational semantics we describe next.

2.4 Operational Semantics

In order to define the operational semantics we should first define the typing system. However, for pedagogical reasons we prefer to follow the reverse order. Let α stands for τ_Δ, and let $\overline{m}{:}\overline{\alpha}$, be the abbreviation of $m_1{:}\alpha_1, \ldots, m_k{:}\alpha_k$, with $k \geq 0$.

Definition 1. Let $m{:}\alpha{\in}R$ if and only if $R \equiv \langle\!\langle R' \mid m{:}\alpha\rangle\!\rangle$ for some R', and define the restriction of a row R according to a label Δ and its negation, denoted by R_{Δ}, and R^{-}_{Δ}, as follows: $R_{\Delta} \triangleq \{m{:}\alpha \in R \mid m \in \Delta\}$, $R^{-}_{\Delta} \triangleq \{m{:}\alpha \in R \mid m \notin \Delta\}$.

The $\lambda_o^{\prec t}$ calculus presents two main challenges. The first is to ensure the conditions of application of extension and overriding: this is ensured by the type system described later on. The second is to give the right type to *self*, which is context dependent: this is done by a suitable typed operational semantics. Here the word "typed" means that some evaluation steps are constrained (i.e. driven) by suitable typing derivations.

Sending message n to the object e is implemented by (i) finding the correct body of method n into the object e, say e_n, (ii) instantiating (i.e. applying) e_n to the higher-order row-abstraction R_n obtained by inspecting (via $\leftarrow\!\!\hookleftarrow$ operator) the object e and (iii) applying the instantiated method body of n to the object itself. One of the subtle points of the operational semantics is that the row-abstraction R_n at point (ii), cannot be determinated at the moment of the message sending (i.e. $e \Leftarrow n$), but must be built during the search, while $\leftarrow\!\!\hookleftarrow$ passes through the various methods. In other words, if we had adopted the following semantics ($\leftarrow\!\!*$ denotes either $\leftarrow\!\!\circ$ or $\leftarrow$):

$$
\begin{array}{lrcll}
(\Leftarrow) & e \Leftarrow n & \overset{ev}{\rightarrow} & (e\leftarrow\!\!\hookleftarrow n)(\lambda t{:}T.R^{-}_{\Delta})e & \quad \text{if } \vdash_A e : \texttt{class}\, t.\langle\!\langle R \mid n{:}\tau_\Delta\rangle\!\rangle \\
(succ) & \langle e_1 \leftarrow\!\!*\ n{=}e_2\rangle \leftarrow\!\!\hookleftarrow n & \overset{ev}{\rightarrow} & e_2 & \\
(next) & \langle e_1 \leftarrow\!\!*\ m{=}e_2\rangle \leftarrow\!\!\hookleftarrow n & \overset{ev}{\rightarrow} & e_1 \leftarrow\!\!\hookleftarrow n, &
\end{array}
$$

then we would not have obtained a correct behavior since this semantics does not take into account methods that are necessary to n but have been added after it (e.g. in case of mutually recursive methods). In particular, subject reduction does not hold[4]. Therefore, we propose the following operational semantic for $\lambda_o^{\prec t}$:

[4] Consider $pt \triangleq \langle\langle x{=}\lambda r{:}T_{[x]}\lambda self{:}\langle\!\langle rt|x{:}int_y\rangle\!\rangle.3\rangle \leftarrow\!\!\circ\ y{=}\ldots\rangle$ of type $\texttt{class}\, t.\langle\!\langle x{:}int_y, y{:}int\rangle\!\rangle$ and verify that the reduction of $pt \Leftarrow x$ by the rules above would not preserve the type.

$$
\begin{array}{lrcll}
(\Leftarrow) & e \Leftarrow n & \overset{ev}{\to} & (e \overset{\lambda t.R^{-}_{\Delta}}{\leftarrow\!\!\!\hookleftarrow} n)e & \text{where } \vdash_A e : \mathtt{class}\, t.\langle\!\langle R \mid n{:}\tau_\Delta \rangle\!\rangle \\
(next_1) & \langle e_1 \leftarrow\!\circ\; m{=}e_2 \rangle \overset{\lambda t.R}{\leftarrow\!\!\!\hookleftarrow} n & \overset{ev}{\to} & e_1 \overset{\lambda t'.R'}{\leftarrow\!\!\!\hookleftarrow} n & \left\{ \begin{array}{l} \text{where } \vdash_A \langle e_1 \leftarrow\!\circ\; m{=}e_2 \rangle : \\ \mathtt{class}\, t'.\langle\!\langle R'' \mid n{:}\tau_\Delta \rangle\!\rangle \;\text{ and} \\ m{:}\alpha \in R''_\Delta \text{ with } R' \equiv \langle\!\langle [t'/t]R \mid m{:}\alpha \rangle\!\rangle \end{array} \right. \\
(next_2) & \langle e_1 \leftarrow\!\ast\; m{=}e_2 \rangle \overset{R}{\leftarrow\!\!\!\hookleftarrow} n & \overset{ev}{\to} & e_1 \overset{R}{\leftarrow\!\!\!\hookleftarrow} n & \text{if } (next_1) \text{ does not apply} \\
(succ) & \langle e_1 \leftarrow\!\ast\; n{=}e_2 \rangle \overset{R}{\leftarrow\!\!\!\hookleftarrow} n & \overset{ev}{\to} & e_2 R & \\
(\beta) & (\lambda a{:}A.B)C & \overset{ev}{\to} & [C/a]B &
\end{array}
$$

where we omit the kind of t in the row-index of $\leftarrow\!\!\!\hookleftarrow$. The rules for error propagation are shown in the Appendix.

We briefly discuss the most important rules, namely the rules $(\Leftarrow)$, $(next_1)$ and $(next_2)$ of the operational semantics. The $(call)$ rule says that sending message n to the object e reduces to apply the search of the method n in question to an higher-order row-abstraction and to the object itself. The row-abstraction passed to the search operator represents all the methods that are introduced after the addition of n but not annotated in the labeled-type of n. The $(next_1)$ rule says that when we are looking for a method body n and we find an object whose outermost method is m and that m is in the labeled-type of n, then we can go "throught" that object but we must record, in the row-index of the search operator, the type of m. Finally, the $(next_2)$ rule apply when the method m we are skipping is not in the labeled-type of n.

As such, the row-abstraction R that is to be passed to the body of n, along the method search will keeps track in its index of those methods used by the body of n and defined after it (rule $(next_1)$) and skips the others (rule $(next_2)$). It follows that the final row R, built during the search of n, contains the types of all the (new) methods which are introduced after the addition of n, together with methods which are not used by n at all (these are put in the index by the rule $(\Leftarrow)$). To deduce this type we use an algorithmic typing system denoted by $\vdash_A$, whose formal presentation is postponed until Section 5.

2.5 Subtyping

The subtyping relation is based on the information given by the labeled-types of methods in rows. We discuss the most important rule, namely the rule which forgets methods in class-types. The full set of rules can be found in the Appendix. We first need the following definition which associates to a set of labeled-types the union of all labels.

Definition 2. The set $\mathcal{L}(\overline{\alpha})$ of methods which occur in the labels of $\overline{\alpha}$ is inductively defined as follows: $\mathcal{L}(\varepsilon) = \{\}$; $\mathcal{L}(\overline{\beta}, \tau_\Delta) = \mathcal{L}(\overline{\beta}) \cup \Delta$.

The $(width_{\preceq})$ rule says that a class-type is a subtype of another class-type if the forgotten methods (i.e. the methods not occurring) in the latter are not in the union of the sets of labels of the remaining methods. The condition $\overline{n} \notin \mathcal{L}(\overline{\alpha})$ formally ensures that the remaining methods do not use the methods $\overline{n}$. Clearly, we can forget groups of mutually recursive methods with this rule. Observe that subtyping is formally forbidden on "open" class-types of the shape, $\mathtt{class}\, t.\langle\!\langle \ldots r \ldots \rangle\!\rangle$. If we would allow subtyping on class-types of that shape, then we would loose the subject reduction property. In fact, our subtyping relation would no longer be closed under substitution of row-variables in class-types.

2.6 Typing Rules

In this subsection we describe only the most interesting typing rules. The full set of rules is described in the Appendix.

The subtyping relation in the previous section is used to type programs through the usual subsumption rule which implies that if $\sigma \preceq \tau$, then an expression e of type σ can be used in any context where an expression of type τ is required. If we use the subsumption rule, then we can obtain judgments of the shape $\Gamma \vdash e : \texttt{class}\, t.R$, where $\Gamma, t{:}T \vdash R : [n]$ even if n is a method that composes e. In this case we say that this rule *forgets* the method n. It is important to remark that, when a method is forgotten in the type of an object, it is like it was never added to the object.

The rule $(obj{-}ext)$ types a method addition producing the new object $\langle e_1 \leftarrow\!\circ\; n{=}e_2 \rangle$. It always adds the method to the syntactic object in case the method is not present or it is present in the object but it was previously forgotten in the type by $(sub_{\preceq})$. The condition $R_{\Delta} = \{\overline{m}{:}\overline{\alpha}\}$ ensures that the $\overline{m}$ methods, needed to type e_2, are present in e_1. But Δ can contain also methods which will be useful to type future bodies of n.

In the $(obj{-}over)$ rule we require that the new method body must have the same type and label as the old one. In the $(obj{-}ext)$ and the $(obj{-}over)$ rules, the method n *uses* some of the methods belonging to the label Δ associated with the labeled-type of n. A difference between the extending and overriding rules in $\lambda_o^{\prec t}$ and those of $\lambda_o^{\prec}$, is that the higher-order row-variable r is explicit in the body of the method we are defining. Having explicit higher-order polymorphism increase the expressiveness of the language, since we can consider method bodies as *first-class*, and so we can type expression such as $(\lambda x{:}\tau.\langle e \leftarrow\!\circ\; n{=}x \rangle)e'$ (see Example 2). This is not possible in $\lambda_o^{\prec}$.

The $(search)$ rule asserts that the type of the extracted method body of n is a function that accept as input argument an object, which will contains the methods $\overline{m}, n$, together with those methods annotated in the the row-index of the search operator.

The $(send)$ rule is a sort of *unfolding* rule; in fact the class-type is a sort of recursive-type.

3 Examples

Example 1 (A Geometric Object). Consider an object *draw* that responds to two messages: *belongs*, that checks whether two integer coordinates superpose to a figure F and *plot*, that, given a point, colors it black or white, depending on the position of the point with respect to the figure F. The method *plot* of *draw* accepts as input both points and colored points (a colored point is a point with an extra field *col* of type *color*). Such a definition is impossible in λ_o, where the only solution would be to write two different *draw* objects, one for colored points the other for points, each one providing a different body for the method *plot*. In fact, for colored points an override instead of an extension should be used:

$$\begin{aligned} draw \;\triangleq\; \langle belongs{=}\lambda r_1{:}T_{[belongs]}.\lambda self_1{:}\texttt{class}\, t.\langle\!\langle r_1 t \mid belongs{:}int{\to}int{\to}bool \rangle\!\rangle. \\ \lambda x{:}int.\lambda y{:}int.(x,y) \in F,\;\; plot{=}\lambda r_2{:}T_{[belongs,plot]}.f \rangle, \end{aligned}$$

where f, the body of *plot*, is:

$$\begin{array}{rl} f \;\triangleq & \lambda self_2{:}\mathtt{class}\,t.\langle\!\langle r_2 t \mid belongs{:}int{\to}int{\to}bool, plot{:}(P{\to}CP)_{belongs}\rangle\!\rangle. \\ & \quad \lambda p{:}P.\text{if } (self_2 \Leftarrow belongs)(p \Leftarrow x)(p \Leftarrow y) \text{ then } \langle p \hookleftarrow\!\circ\; col = \lambda r_3{:}T_{[col]}.set\_black\rangle \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{else } \langle p \hookleftarrow\!\circ\; col = \lambda r_4{:}T_{[col]}.set\_white\rangle, \end{array}$$

with *set_black* and *set_white* denoting respectively, $\lambda self_3{:}\mathtt{class}\,t.\langle\!\langle r_3 t \mid col{:}color\rangle\!\rangle.black$ and $\lambda self_4{:}\mathtt{class}\,t.\langle\!\langle r_4 t \mid col{:}color\rangle\!\rangle.white$, $P \equiv \mathtt{class}\,t.\langle\!\langle x{:}int, y{:}int, mv{:}(int{\to}int{\to}t)_{x,y}\rangle\!\rangle$, and $CP \equiv \mathtt{class}\,t.\langle\!\langle x{:}int, y{:}int, mv{:}(int{\to}int{\to}t)_{x,y}, col{:}color\rangle\!\rangle$. Then we can derive:

$$\varepsilon \vdash draw : \mathtt{class}\,t.\langle\!\langle belongs{:}int{\to}int{\to}bool, plot{:}(P{\to}CP)_{belongs}\rangle\!\rangle.$$

Example 2 (First-Class Methods). Let *point* be a geometric point, and consider the program $(\lambda x{:}(\forall r{:}T_{[col]}.\sigma{\to}color).\langle point \hookleftarrow\!\circ\; col{=}x\rangle)(\lambda r{:}T_{[col]}.\lambda self{:}\sigma.black)$, where $\sigma \equiv \mathtt{class}\,t.\langle\!\langle rt \mid col{:}color\rangle\!\rangle$. This program is parametric in the body of the *col* method to be added to the object *point*. It is easy to verify that *type-erasure* of this program, i.e. $(\lambda x.\langle point \hookleftarrow\!\circ\; col{=}x\rangle)(\lambda self.black)$, cannot be typed in $\lambda_o^{\prec}$.

Excursus. The terms above may appear too complicated, especially in their type annotations, for a realistic use. However, from a pragmatic point of view, it is not diffucult to build an equivalent functional language, where objects can be easily manipulated and type-checked and where much less type decoration is required. Indeed, the type information contained in a row-abstraction can be inferred by the fixed part of the *my-type* decoration. For example, in such a kernel language, the object *point* of Section 2.3 might be written as follows:

```
<x  = fun(self with x:int) is 0,
 y  = fun(self with y:int) is 0,
 mv = fun(self with (x:int,y:int,mv:int->int->Mytype),dx:int,dy:int) is
              <<self <- x = fun(self1 with x:int) self.x+dx>
                     <- y = fun(self1 with y:int) self.y+dy>
>,
```

where `Mytype` denotes the type of *self*. It is easy too see that the “program” above contains all the type information needed to check its type safety.

4 Properties of the Type System

In this section, we show the properties satisfied by our calculus. Because of lack of space, all proofs and technical lemmas are omitted.

To prove subject reduction, we need a series of preliminary results that help isolate some interesting properties of the type system. Throughout the rest of the paper, we will also appeal to a notion of *normal form* for our type derivations that allows us to simplify proofs of theorems and lemmas. We refer the reader to [7] for the definition of such normal form and to [2] for its adaptation to the case of labeled-types. The notion of normal form is well defined for our system. The only difference here is that, since types occur in expressions, a derivation is in normal form only if all type-decorations are also in β-normal form.

The theorem of subject reduction assures that well-typed expressions reduce only to well-typed expressions. Since *err* is not typable, then this theorem ensures that static type-checking avoid run-time type errors.

Theorem 3 (Subject Reduction and Type Soundness).

1. *If $\Gamma \vdash e : \tau$ and $e \overset{ev}{\twoheadrightarrow} e'$, then $\Gamma \vdash e' : \tau$.*
2. *If $\varepsilon \vdash e : \tau$, then the evaluation of e cannot produce err, i.e. $e \overset{ev}{\not\twoheadrightarrow}$err.*

The introduction of explicit universal quantification increase the expressivity of our typed lambda calculus of objects. More precisely, the first result that we prove is that there exists an *erasing function* $\mathcal{E}$ from typed to untyped terms, such that every expression derivable in $\lambda_o^{\prec}$ can be derivable as the erasure of some typed expression in $\lambda_o^{\prec t}$. This result shows that $\lambda_o^{\prec t}$ is at least as expressive as $\lambda_o^{\prec}$. The erasing function $\mathcal{E}$, whose definition can be found in the Appendix, simply erases the type annotations on the abstracted variables, and eliminates the expression-row applications.

Lemma 4 (Erasing). *Let $\overset{ev}{\rightarrow}_t$ be the reduction for $\lambda_o^{\prec t}$, and $\overset{ev}{\rightarrow}_u$ the reduction for $\lambda_o^{\prec}$.*

1. *$\mathcal{E}([B/b]A) \equiv [\mathcal{E}(B)/b]\mathcal{E}(A)$.*
2. *If $A \overset{ev}{\rightarrow}_t B$, then either $\mathcal{E}(A) \overset{ev}{\rightarrow}_u \mathcal{E}(B)$, or $\mathcal{E}(A) \equiv \mathcal{E}(B)$.*

Theorem 5 (Completeness of $\lambda_o^{\prec t}$ w.r.t. $\lambda_o^{\prec}$). *Let $\Gamma \vdash e : \tau$ in $\lambda_o^{\prec}$. Then:*

1. *There exists a typed (legal) context Γ_t, a typed expression e_t, and a typed type τ_t, satisfying $\mathcal{E}(\Gamma_t) \equiv \Gamma$, $\mathcal{E}(e_t) \equiv e$ and $\mathcal{E}(\tau_t) \equiv \tau$, such that $\Gamma_t \vdash_t e_t : \tau_t$ in $\lambda_o^{\prec t}$.*
2. *For every typed (legal) context Γ_t and typed type τ_t, satisfying $\mathcal{E}(\Gamma_t) \equiv \Gamma$ and $\mathcal{E}(\tau_t) \equiv \tau$, there exists a typed expression e_t, such that $\Gamma_t \vdash_t e_t : \tau_t$ in $\lambda_o^{\prec t}$, and $\mathcal{E}(e_t) \equiv e$.*

Lemma 6. *Let $\Gamma \vdash e : \tau$ in $\lambda_o^{\prec t}$. If $\mathcal{E}(e) \overset{ev}{\twoheadrightarrow}_u e'$ then there exists e'' such that $e' \equiv \mathcal{E}(e'')$ and $e \overset{ev}{\twoheadrightarrow}_t e''$.*

The last lemma ensures that the computation of an expression of the typed calculus can be performed on its erasure, since the result will be the same (modulo erasures): in other words, after static type-checking, the expressions can be compiled into their erasures.

Corollary 7 (Untyped Soundness). *If $\varepsilon \vdash e : \tau$, then the untyped evaluation of $\mathcal{E}(e)$ cannot produce err, i.e. $\mathcal{E}(e) \overset{ev}{\not\twoheadrightarrow}_u$ err.*

5 Algorithmic Version of the Calculus

The typing rules of $\lambda_o^{\prec t}$ do not specify a deterministic typing algorithm. A set of rules specify a deterministic typing algorithm if they are *syntax-directed*, and, moreover, if every rule satisfies the *subformula property*, i.e. all the formulas appearing in the premise of a rule are subformulas of those appearing in the conclusion (see [4] for general definitions). The $(sub_{\preceq})$ rule is not deterministic, because that uses the subtyping relation $\preceq$ which has not a deterministic specification. The problem comes from the rules $(refl_{\preceq})$ and $(trans_{\preceq})$, that are not syntax-directed. Furthermore, the $(trans_{\preceq})$ rule does not satisfy the subformula property.

The next lemma proves that the $(trans_{\preceq})$ and $(refl_{\preceq})$ rules are useless since they can be derived from the remaining ones.

Lemma 8. *A subtyping judgment is provable in the typing system containing the rules $(trans_{\preceq})$, and $(refl_{\preceq})$, if and only if it is provable without them.*

5.1 Typing Algorithm

The subsumption rule is not the only problematic rule. In this calculus, we have higher-order bounded polymorphism on kinds: the $(rlab)$ rule can also be considered as a subsumption rule on kinds. Thus, in order to obtain an algorithmic presentation for $\lambda_o^{\prec t}$, we need to eliminate both the $(sub_{\preceq})$ and $(rlab)$ rules, and to modify the $(eapp)$, $(obj{-}ext)$, $(search)$, and $(erapp)$ rules. Let $\Gamma \vdash_A e : \sigma \preceq \tau$ denote $\Gamma \vdash_A e : \sigma$ and $\Gamma \vdash_A \sigma \preceq \tau$.

Definition 9. The algorithmic system, $\vdash_A$, is defined from $\vdash$ system in which:
1. remove the $(trans_{\preceq})$, $(refl_{\preceq})$, $(sub_{\preceq})$, and $(rlab)$ rules;
2. modify the $(eapp)$, $(obj{-}ext)$, $(search)$, and $(erapp)$ rules as follows:

$$\frac{\Gamma \vdash_A e_1 : \sigma \to \tau \quad \Gamma \vdash_A e_2 : \sigma' \quad \Gamma \vdash_A \sigma' \preceq \sigma}{\Gamma \vdash_A e_1 e_2 : \tau} \quad (eapp)$$

$$\frac{\begin{array}{l}\Gamma \vdash_A e_1 : \mathtt{class}\, t.R' \preceq \mathtt{class}\, t.R \quad \Gamma, t{:}T \vdash_A R : [\overline{p}] \quad R_\Delta = \{\overline{m}{:}\overline{\alpha}\} \\ \Gamma \vdash_A e_2 : \forall r{:}T_{[\overline{m},n]}.[\mathtt{class}\, t.\langle\!\langle rt \mid \overline{m}{:}\overline{\alpha}, n{:}\tau_\Delta \rangle\!\rangle / t](t \to \tau) \quad r \text{ not in } \tau \quad n \in \{\overline{p}\}\end{array}}{\Gamma \vdash_A \langle e_1 \leftarrow\!\circ\; n{=}e_2 \rangle : \mathtt{class}\, t.\langle\!\langle R \mid n{:}\tau_\Delta \rangle\!\rangle} \quad (obj{-}ext)$$

$$\frac{\Gamma \vdash e : \mathtt{class}\, t.\langle\!\langle R' \mid n{:}\tau_\Delta \rangle\!\rangle \quad R'_\Delta = \{\overline{m}{:}\overline{\alpha}\} \quad \Gamma \vdash R : T \to [\overline{p}] \quad \{\overline{m}, n\} \subseteq \{\overline{p}\}}{\Gamma \vdash e \overset{R}{\longleftrightarrow} n : [\mathtt{class}\, t.\langle\!\langle Rt \mid \overline{m}{:}\overline{\alpha}, n{:}\tau_\Delta \rangle\!\rangle / t](t \to \tau)} \quad (search)$$

$$\frac{\Gamma \vdash_A e : \forall r{:}T^p \to [\overline{m}].\tau \quad \Gamma \vdash_A R : T^p \to [\overline{n}] \quad \{\overline{m}\} \subseteq \{\overline{n}\}}{\Gamma \vdash_A eR : [R/r]\tau} \quad (erapp)$$

Lemma 10.
1. **(Uniqueness of Kinding)** *If $\Gamma \vdash_A R : \kappa$, then κ is unique.*
2. **(Uniqueness of Typing)** *If $\Gamma \vdash_A e : \tau$, then τ is unique.*
3. **(Soundness)** *If $\Gamma \vdash_A e : \tau$, then $\Gamma \vdash e : \tau$.*

The main theorem of this section states the completeness of the typing algorithm, namely that every expression that is typeable by $\vdash$ is typeable by $\vdash_A$.

Theorem 11 (Completeness). *If $\Gamma \vdash e : \tau$, then $\Gamma \vdash_A e : \sigma$ and $\Gamma \vdash_A \sigma \preceq \tau$.*

Corollary 12 (Minimum Type). *If $\Gamma \vdash_A e : \tau$, then $\tau = min_{\preceq}\{\tau' \mid \Gamma \vdash e : \tau'\}$.*

Proposition 13. *The $\vdash_A$ type system defines a terminating algorithm.*

We conclude this section with the decidability results.

Theorem 14. *Given a closed expression e, and a closed type τ, the following propositions are decidable:*
1. **(Type Checking)** *the judgment $\varepsilon \vdash_A e : \tau$ is derivable.*
2. **(Type Reconstruction)** *there exists a type τ', such that $\varepsilon \vdash_A e : \tau'$.*

6 Related Work

In [8], an interesting solution for adding subtyping to λ_o is proposed. The main novelty is the introduction of a new type, the *pro-type*, denoted by $\mathtt{pro}\, t.R$, which can be intended as the $\mathtt{class}\, t.R$ type of [7]. If we can assign a pro-type to an object, then we can add new methods or override existing ones. At this level, only trivial subtyping is possible. Then we can "convert" the object into a different kind of object where methods cannot be altered —i.e. the only operation on objects is message send— by "sealing" a pro-type into a real object-type. Even if from the outside of the object the only operation is message send, the internal methods can override other methods of their host object. Preventing from the outside extension and override gives (*mytype* covariant) "width" and "depth" subtyping. This solution gives a depth knowledge about how classes can be understood, encapsulated and implemented, but there are programs which can be typed in [2] and cannot be typed in [8] and vice versa.

An alternative approach to object-based languages is the Theory of Primitive Object of Abadi and Cardelli [1] where the only operations allowed are method override and message send (extension is missing). This calculus, based on fixed-size objects, instead of open-ended extensible objects, features (*mytype* covariant) "width" subtyping, and the type of *self* is modeled via a second order encoding. Recently, the first author has developed a conservative extension of the Abadi and Cardelli calculus of objects which provide for object extension and "width" subtyping [9].

Finally, we mentioned the possibility of integrating our calculus with λ&-calculus of [5], by allowing *multi-methods*, and which is illustrated in a companion paper [6].

Acknowledgements. The authors wish to thank Mariangiola Dezani-Ciancaglini for her precious technical support and encouragement and the anonymous referees for their comments and suggestions.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
2. 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.
3. G. Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, 1995.
4. G. Castagna. *Object-Oriented Programming: A Unified Foundation*. Progress in Theoretical Computer Science. Birkäuser, Boston, 1996.
5. G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. *Information and Computation*, 117(1):115–135, 1995.
6. G. Castagna and L. Liquori. Multi-methods in delegation-based object-oriented languages. Manuscript, 1996.
7. 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.
8. 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.
9. L. Liquori. An Extended Theory of Primitive Objects. Technical Report CS-23-96, Computer Science Department, Turin University, Italy, 1996.

10. J. C. Mitchell. Toward a Typed Foundation for Method Specialization and Inheritance. In *Proc. of POPL*, pages 109–124. The ACM Press, 1990.
11. J. Palsberg and T. Jim. Type Inference for Simple Object Types is NP-Complete. Manuscript, 1995.

Appendix

1: Operational Semantics (error-propagation rules)

$$
\begin{array}{lrcl@{\qquad}lrcl}
(fail_1) & \langle\rangle \overset{R}{\leftarrow\!\!\hookleftarrow} n & \overset{ev}{\to} & err & (fail_2) & \lambda a{:}A.B \overset{R}{\leftarrow\!\!\hookleftarrow} n & \overset{ev}{\to} & err \\
(err \leftarrow\!\ast\ ) & \langle err \leftarrow\!\ast\ m = e\rangle & \overset{ev}{\to} & err & (err \leftarrow\!\!\hookleftarrow) & err \leftarrow\!\!\hookleftarrow n & \overset{ev}{\to} & err \\
(err\ abs) & \lambda x{:}\tau.err & \overset{ev}{\to} & err & (err\ appl) & err\ e & \overset{ev}{\to} & err
\end{array}
$$

2: Erasing

$$
\begin{array}{rcl@{\qquad}rcl}
\mathcal{E}(eR) & = & \mathcal{E}(e) & \mathcal{E}(\forall r{:}\kappa.\tau) & = & \mathcal{E}(\tau) \\
\mathcal{E}(\lambda x{:}\tau.e) & = & \lambda x.\mathcal{E}(e) & \mathcal{E}(\langle\!\langle R \mid n{:}\tau_\Delta \rangle\!\rangle) & = & \langle\!\langle \mathcal{E}(R) \mid n{:}\mathcal{E}(\tau)_\Delta \rangle\!\rangle \\
\mathcal{E}(\lambda r{:}\kappa.e) & = & \mathcal{E}(e) & \mathcal{E}(\lambda t{:}T.R) & = & \lambda t.\mathcal{E}(R) \\
\mathcal{E}(e \overset{R}{\leftarrow\!\!\hookleftarrow}\ n) & = & \mathcal{E}(e) \leftarrow\!\!\hookleftarrow n & & &
\end{array}
$$

Otherwise, $\mathcal{E}$ is compositional or the identity.

3: Typing Rules

Let a, A, B, ... range over terms, and let $T^p \to [\overline{m}]$ stand for $\underbrace{T \to \ldots \to T}_{p} \to [\overline{m}]$.

Valid Signatures and Axioms

$$
\frac{}{\emptyset\ sig}\,(ax_\Sigma) \qquad \frac{}{\varepsilon \vdash \ast}\,(ax) \qquad \frac{\Sigma\ sig \quad \vdash A \ast B \quad a \notin \mathcal{D}om\,(\Sigma)}{\Sigma, a{:}A\ sig}\,(var_\Sigma)
$$ [5]

General Rules

$$
\frac{\Gamma \vdash \ast \quad a{:}A \in \Sigma}{\Gamma \vdash a : A}\,(pj_\Sigma) \qquad \frac{\Gamma \vdash \ast \quad a{:}A \in \Gamma}{\Gamma \vdash a : A}\,(pj)
$$

$$
\frac{\Gamma \vdash A : B \quad \Gamma, \Gamma' \vdash \ast}{\Gamma, \Gamma' \vdash A : B}\,(wk) \qquad \frac{\Gamma \vdash A : B \quad a \notin Dom(\Gamma)}{\Gamma, a{:}A \vdash \ast}\,(var)
$$

Rules for Types

$$
\frac{\Gamma \vdash \ast \quad t \notin Dom(\Gamma)}{\Gamma, t{:}T \vdash \ast}\,(tvar) \qquad \frac{\Gamma \vdash \tau_1 : T \quad \Gamma \vdash \tau_2 : T}{\Gamma \vdash \tau_1 \to \tau_2 : T}\,(tarr) \qquad \frac{\Gamma, t{:}T \vdash R : [\overline{m}]}{\Gamma \vdash \texttt{class}\ t.R : T}\,(class)
$$

[5] Where $A \ast B$ denotes either $c : \iota$, $\iota : T$ or $\iota_1 \leq \iota_2$, and c is a constant expression.

Type and Row Equality

$$\frac{\Gamma \vdash \tau : T \quad \tau \to_\beta \tau'}{\Gamma \vdash \tau' : T}\,(t\beta) \qquad \frac{\begin{array}{l}\Gamma \vdash e : \tau\\ \Gamma \vdash \tau' : T \quad \tau =_\beta \tau'\end{array}}{\Gamma \vdash e : \tau'}\,(teq) \qquad \frac{\Gamma \vdash R : \kappa \quad R \to_\beta R'}{\Gamma \vdash R' : \kappa}\,(r\beta)$$

Rules for Rows

$$\frac{\Gamma \vdash * \quad \bar{l}\ \mathit{fixed}}{\Gamma \vdash \langle\!\langle\rangle\!\rangle : [\bar{l}]}\,(erow) \qquad \frac{\Gamma \vdash * \quad r \notin \mathit{Dom}(\Gamma)}{\Gamma, r{:}T^p{\to}[\overline{m}] \vdash *}\,(rvar)$$

$$\frac{\Gamma, t{:}T \vdash R : T^p{\to}[\overline{m}]}{\Gamma \vdash \lambda t{:}T.R : T^{p+1}{\to}[\overline{m}]}\,(rabs) \qquad \frac{\Gamma \vdash R : T{\to}\kappa \quad \Gamma \vdash \tau : T}{\Gamma \vdash R\tau : \kappa}\,(rapp)$$

$$\frac{\Gamma \vdash R : [\overline{m}, n] \quad \Gamma \vdash \tau : T}{\Gamma \vdash \langle\!\langle R \mid n{:}\tau_\Delta\rangle\!\rangle : [\overline{m}]}\,(rext) \qquad \frac{\Gamma \vdash R : T^p{\to}[\overline{m}] \quad \{\overline{n}\} \subseteq \{\overline{m}\}}{\Gamma \vdash R : T^p{\to}[\overline{n}]}\,(rlab)$$

Rules for Expressions

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

$$\frac{\Gamma, r{:}\kappa \vdash e : \tau}{\Gamma \vdash \lambda r{:}\kappa.e : \forall r{:}\kappa.\tau}\,(erabs) \qquad \frac{\Gamma \vdash e : \forall r{:}\kappa.\tau \quad \Gamma \vdash R : \kappa}{\Gamma \vdash eR : [R/r]\tau}\,(erapp)$$

$$\frac{\Gamma \vdash e : \sigma \quad \Gamma \vdash \sigma \preceq \tau}{\Gamma \vdash e : \tau}\,(sub_{\preceq}) \qquad \frac{\Gamma \vdash e : \mathtt{class}\,t.\langle\!\langle R \mid n{:}\tau_\Delta\rangle\!\rangle}{\Gamma \vdash e \Leftarrow n : [\mathtt{class}\,t.\langle\!\langle R \mid n{:}\tau_\Delta\rangle\!\rangle/t]\tau}\,(send)$$

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

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \mathtt{class}\,t.\langle\!\langle R \mid n{:}\tau_\Delta\rangle\!\rangle \qquad R_\Delta = \{\overline{m}{:}\overline{\alpha}\}\\ \Gamma \vdash e_2 : \forall r{:}T_{[\overline{m},n]}.[\mathtt{class}\,t.\langle\!\langle rt \mid \overline{m}{:}\overline{\alpha}, n{:}\tau_\Delta\rangle\!\rangle/t](t{\to}\tau)\end{array}}{\Gamma \vdash \langle e_1 \leftarrow n{=}e_2\rangle : \mathtt{class}\,t.\langle\!\langle R \mid n{:}\tau_\Delta\rangle\!\rangle}\,(obj{-}over)$$

$$\frac{\Gamma \vdash e : \mathtt{class}\,t.\langle\!\langle R' \mid n{:}\tau_\Delta\rangle\!\rangle \quad R'_\Delta = \{\overline{m}{:}\overline{\alpha}\} \quad \Gamma \vdash R : T_{[\overline{m},n]}}{\Gamma \vdash e \overset{R}{\longleftarrow\!\hookleftarrow} n : [\mathtt{class}\,t.\langle\!\langle Rt \mid \overline{m}{:}\overline{\alpha}, n{:}\tau_\Delta\rangle\!\rangle/t](t{\to}\tau)}\,(search)$$

Rules of Subtyping

$$\frac{\Gamma \vdash \sigma : T}{\Gamma \vdash \sigma \preceq \sigma}\,(refl_{\preceq}) \qquad \frac{\Gamma \vdash \sigma \preceq \tau \quad \Gamma \vdash \tau \preceq \rho}{\Gamma \vdash \sigma \preceq \rho}\,(trans_{\preceq})$$

$$\frac{\Gamma \vdash \sigma' \preceq \sigma \quad \Gamma \vdash \tau \preceq \tau'}{\Gamma \vdash \sigma{\to}\tau \preceq \sigma'{\to}\tau'}\,(arrow_{\preceq}) \qquad \frac{\Gamma \vdash \mathtt{class}\,t.\langle\!\langle \overline{m}{:}\overline{\alpha}, \overline{n}{:}\overline{\beta}\rangle\!\rangle : T \quad \overline{n} \notin \mathcal{L}(\overline{\alpha})}{\Gamma \vdash \mathtt{class}\,t.\langle\!\langle \overline{m}{:}\overline{\alpha}, \overline{n}{:}\overline{\beta}\rangle\!\rangle \preceq \mathtt{class}\,t.\langle\!\langle \overline{m}{:}\overline{\alpha}\rangle\!\rangle}\,(width_{\preceq})$$