A protoype-based approach to object reclassification

Alberto Ciaffaglione, Pietro Di Gianantonio, Furio Honsell, Luigi Liquori

Submitted to Journal of Object Technology
Published by AITO — Association Internationale pour les Technologies Objets
http://www.jot.fm/

A protoype-based approach to object reclassification

Alberto Ciaffaglione[a] Pietro Di Gianantonio[a] Furio Honsell[a]
Luigi Liquori[b]

a. University of Udine, Italy
b. INRIA Sophia Antipolis Méditerranée, France

Abstract We investigate, in the context of *functional prototype-based languages*, a calculus of objects which might extend themselves upon receiving a message, a possibility referred to by Cardelli as a *self-inflicted* operation. We present a sound type system for this calculus which guarantees that evaluating a well-typed expression will never yield a `message-not-found` runtime error. The resulting calculus is an attempt towards the definition of a language combining the safety advantage of static type check with the flexibility normally found in dynamically typed languages.

Keywords Prototype-based calculi, static typing, object reclassification

1 Introduction

Object calculi and languages can be divided in the two main categories of class-based and prototype-based (a.k.a. object-based) ones. The latter, whose best-known example is `JavaScript`, provide the programmer with a greater flexibility compared to those classed-based, e.g. the possibility of changing at runtime the behaviour of objects, by modifying or adding methods. Although such a flexibility is normally payed by the lack of static type systems, this is not necessarily the case, as it is possible to define a statically typed, prototype-based language. One example in this direction is the *Lambda Calculus of Objects* ($\lambda\mathcal{O}bj$), introduced by Fisher, Honsell, and Mitchell [FHM94] as a first solid foundation for the prototyped-based paradigm.

$\lambda\mathcal{O}bj$ is a lambda calculus extended with object primitives, where a new object may be created by modifying or extending an existing *prototype*. The new object thereby inherits properties from the original one in a controlled manner. Objects can be viewed as lists of pairs *(method name, method body)* where the method body is (or reduces to) a lambda abstraction whose first formal parameter is always `self` (or `this` in $C^{++}, Java$).

The type assignment system of $\lambda\mathcal{O}bj$ is set up so as to prevent the unfortunate `message-not-found` runtime error. Types of methods are allowed to be *specialized* to

Alberto Ciaffaglione, Pietro Di Gianantonio, Furio Honsell, Luigi Liquori. *A protoype-based approach to object reclassification*. Licensed under . In Submitted to Journal of Object Technology, pages 1–39.
Available at http://www.jot.fm/

the type of the inheriting objects. This feature, usually referred to as "Mytype method specialization", reinterprets the symbol `self` in the type of the inheriting object. This high mutability of method bodies is accommodated in the type system via an implicit form of *higher-order polymorphism*, inspired by the the work of Wand on extensible records [Wan87].

The calculus $\lambda\mathcal{O}bj$ has spurred an intense research in type assignment systems for object calculi. Several calculi inspired by $\lambda\mathcal{O}bj$, which accommodate various extra features such as incomplete objects, subtyping, encapsulation, imperative features, have appeared soon afterwards (see *e.g.* [FM95, BL95, BBDL97, FM98, BF98]).

More specifically, $\lambda\mathcal{O}bj$ supports two operations which may change the shape of an object: *method addition* and *method override*. The operational semantics of the calculus allows method bodies in objects to modify their own `self`, a powerful capability referred to by Cardelli as a *self-inflicted* operation [Car95].

Example 1.1 Consider the method `set_x` *belonging to a* `point` *object with an* `x` *field:*

$$\begin{array}{lcll} \texttt{point} & \triangleq & \langle \texttt{x} & = & \lambda\texttt{self}.1, \\ & & \texttt{set\_x} & = & \lambda\texttt{self}.\lambda\texttt{v}.\langle \texttt{self} \leftarrow\!\!\!- \texttt{x} = \lambda\texttt{self}.\texttt{v}\rangle, \\ & & \ldots\rangle. & & \end{array}$$

When `set_x` *is sent to* `point` *with an argument* `3`*, written as* $\texttt{point} \Leftarrow \texttt{set\_x}(3)$*, the result is a new object where the* `x` *field has been set (*i.e. *overridden) to* `3`*. Notice the self-inflicted operation of object override (*i.e. $\leftarrow\!\!\!-$*) performed by the* `set_x` *method.*

However, in all the type systems for calculi of objects, both those derived from $\lambda\mathcal{O}bj$ and those derived from Abadi and Cardelli's foundational *Object Calculus* [AC96], the type system prevents the possibility for a method to self-inflict an extension to the host object. We feel that this is an unpleasant limitation if the *message-passing paradigm* is to be taken in full generality. Moreover, in $\lambda\mathcal{O}bj$ this limitation appears arbitrary, given that the operational semantics supports without difficulty self-inflicted extension methods.

There are plenty of situations, both in programming and in real life, where it would be convenient to have objects which modify their interface upon an execution of a message. Consider for instance the following situations.

- The process of *learning* could be easily modeled using an object which can react to the "teacher's message" by extending its capability of performing, in the future, a new task in response to a new request from the environment (an old dog could appear to learn new tricks if in his youth it had been taught a "self-extension" trick).
- The process of "vaccination" against the virus $\mathcal{X}$ can be viewed as the act of extending the capability of the immune system of producing, in the future, a new kind of "$\mathcal{X}$-antibodies" upon receiving the message that an $\mathcal{X}$-infection is in progress.
- In standard typed class-based languages we can modify the structure of the class only statically. If we need to add a new method to an instance of a class we are forced to *re*-compile the class and to make the modification needlessly available to *all the class-instances*, thereby wasting memory. If a class had a self-extension method, only the instances of the class which have dynamically executed this method would allocate new memory, without the need of any re-compilation. As

a consequence, many sub-class declarations could be easily explained away if suitable self-extension methods in the parent class were available.

- *Down-casting* could be smoothly implementable on objects with self-extension methods. For example, the following expression could be made to type-check:

$$\texttt{col\_point} \Leftarrow \texttt{equal}\,(\texttt{point} \Leftarrow \texttt{add\_set\_col}\,(\texttt{black})),$$

  where `add_set_col` is a self-extension method of `point`, and `equal` is the name of the standard binary "equality" method.

- Self-extension appears to be strictly related to object evolution and object reclassification (see Sections 7 and 8), two features which are required in the software world by domains such as e.g. banking, GUI development, and games.

Actually, the possibility of modifying objects at runtime is already available in dynamically typed languages such as `Smalltalk` (via the `become` method), `Python` (by modifying the `_class_` attribute), and `Ruby`. On the other hand, the self-extension itself is present, and used, in the prototype-based `JavaScript` language.

In such a scenario, the objective of this paper is to introduce the prototype-based $\lambda\mathcal{O}bj+$, a lambda calculus of objects in the style of $\lambda\mathcal{O}bj$, together with a type assignment system which allows self-inflicted extension still catching *statically* the `message-not-found` runtime error. This system can be further extended to accommodate other "subtyping" features; by way of example we will present a width-subtyping relation that permits sound method override and a limited form of object extension.

The research presented in this article belongs to a series of similar investigations [Zha10, CHJ12, Zha12], whose aim is to define more and more powerful type assignment systems, capable to statically type check larger and larger fragments of a prototype-based, dynamically typed language like `JavaScript`. The ultimate goal is the definition of a language combining the safety advantage of static type check with the flexibility normally found in dynamically typed languages.

Self-inflicted Extension

To enable the $\lambda\mathcal{O}bj+$ calculus to perform self-inflicted extensions, two modifications of the system in [FHM94] are necessary. The first is, in effect, a simplification of the original syntax of the language. The second is much more substantial and it involves the type discipline.

As far as the syntax of the language is concerned, we are forced to *unify* into a *single* operator, denoted by $\prec\!\!\leftarrow$, the two original object operators of $\lambda\mathcal{O}bj$, *i.e.* object extension ($\leftarrow\!\!+$) and object override ($\leftarrow\!\!-$). This is due to the fact that, when iterating the execution of a self-extension method, only the first time we have a genuine object extension, while the second time we have just a simple object override.

Example 1.2 Consider the method `add_set_col`*, that adds and sets a* `col` *field:*

$$\texttt{point} \triangleq \langle \texttt{add\_set\_col} = \lambda\texttt{self}.\lambda\texttt{v}.\langle \texttt{self} \prec\!\!\leftarrow \texttt{col} = \lambda\texttt{self}.\texttt{v}\rangle, \ldots\rangle.$$

When `add_set_col` *is sent to* `point` *with argument* `white`, i.e. `point` $\Leftarrow$ `add_set_col(white)`*, the result is a new object* `col_point` *where the* `col` *field has been added to* `point` *and set to* `white`*. If* `add_set_col` *is sent twice to* `point`*, as follows:*

$$\texttt{col\_point} \Leftarrow \texttt{add\_set\_col}\,(\texttt{black})$$

then, since the field `col` *is already present in* `col_point`*, the field* `col` *is overridden with* `black`.

As far as types are concerned, we add two new kinds of object-types, namely $\tau \lhd \mathtt{m}$, which can be seen as the semantic counterpart of the syntactic one $\langle e_1 \prec\!\!\!\leftarrow \mathtt{m} = e_2 \rangle$, and $\mathtt{pro}\, t.\langle\!\langle R \rangle\!\rangle \lhd \mathtt{m}_1 \ldots \lhd \mathtt{m}_n$, where R is a *row* containing a superset of the methods and related types present in the object-type $\mathtt{class}\, t.R'$ of [FHM94].

If the type $\mathtt{pro}\, t.\langle\!\langle R \rangle\!\rangle \lhd \mathtt{m}_1 \ldots \lhd \mathtt{m}_n$ is assigned to an object e, then e can respond to all the methods $\mathtt{m}_1 \ldots \mathtt{m}_n$. The list R contains the same methods $\mathtt{m}_1 \ldots \mathtt{m}_n$ together with the corresponding types; moreover, R may contain some *reserved* methods, i.e. methods that can be added to e either by ordinary object-extension or by a method in R which performs a self-inflicted extension.

Example 1.3 Consider an object e *which is assigned the type* $\mathtt{pro}\, t.\langle\!\langle \mathtt{m} : t \lhd \mathtt{n},\ \mathtt{n} : \mathtt{int} \rangle\!\rangle \lhd \mathtt{m}$. *Then* $e \Leftarrow \mathtt{m}$ *produces the effect of adding the field* `n` *to the interface of* e*, and of updating the type of* e *to* $\mathtt{pro}\, t.\langle\!\langle \mathtt{m} : t \lhd \mathtt{n},\ \mathtt{n} : \mathtt{int} \rangle\!\rangle \lhd \mathtt{m} \lhd \mathtt{n}$.

The list of reserved methods in an object-type is crucial to enforce the consistency of the type assignment system. Consider e.g. an object containing two methods, `add_n1`, and `add_n2`, each of them self-inflicting the extension of a new method `n`. The type assignment system has to carry enough information so as to enforce that the same type will be assigned to `n` whatever self-inflicted extension has been executed.

The typing system that we will introduce ensures that we can always dynamically add new fresh methods for `pro`-types, thus leaving intact the original "philosophy" of rapid prototyping, peculiar to object calculi.

To model specialization of inherited methods, we use the notion of *matching* or type extension, originally introduced by Bruce [Bru94], and later applied to the Object Calculus [AC96] and to $\lambda\mathcal{O}bj$ [BB99]. At the price of a little more mathematical overhead, we could have used also the implicit higher-order polymorphism of [FHM94].

Object Subsumption.

As it is well-known, see *e.g.* [AC96, FM94], the introduction of a subsumption relation over object-types makes the type system unsound. In particular, width-subtyping clashes with object extension, and depth-subtyping clashes with object override. In fact, on `pro`-types no subtyping is possible. In order to accommodate subtyping, we add another kind of object-type, *viz.* $\mathtt{obj}\, t.\langle\!\langle R \rangle\!\rangle \lhd \mathtt{m}_1 \ldots \lhd \mathtt{m}_n$, which behaves like $\mathtt{pro}\, t.\langle\!\langle R \rangle\!\rangle \lhd \mathtt{m}_1 \ldots \lhd \mathtt{m}_n$ except that it can be assigned to objects which can be extended only by making longer the list $\mathtt{m}_1 \ldots \mathtt{m}_n$ (by means of the reserved methods that appear in R). On `obj`-types a (covariant) width-subtyping is permitted[1].

Synopsis. The present paper is organized as follows. In Section 2 we introduce the calculus $\lambda\mathcal{O}bj+$, its small-step operational semantics, and some intuitive examples to illustrate the idea of self-inflicted object extension. In Section 3 we define the type system for $\lambda\mathcal{O}bj+$ and we discuss in detail the intended meaning of the most interesting rules. In Section 4 we show how our type system is compatible with a width-subtyping relation. A collection of examples is presented in Section 5. In Section 6 we state our soundness result, namely that every closed and well-typed expression will not produce wrong results. Section 7 is devoted to workout an example, to illustrate the potential of the self-inflicted extension mechanism as a runtime feature, in connection

[1] The `pro` and `obj` terminology is borrowed from Fisher and Mitchell [FM95, FM98].

$$
\begin{array}{lll}
(Beta) & (\lambda x.e_1)\, e_2 & \stackrel{ev}{\rightarrow} \quad [e_2/x]e_1 \\
(Select) & e \Leftarrow \mathtt{m} & \stackrel{ev}{\rightarrow} \quad Sel(e,\, \mathtt{m},\, \lambda x.\, x) \\
(Success) & Sel(\langle e_1 \prec\!\!\!\leftarrow \mathtt{m} = e_2\rangle,\, \mathtt{m},\, e) & \stackrel{ev}{\rightarrow} \quad e_2\, (e\, \langle e_1 \prec\!\!\!\leftarrow \mathtt{m} = e_2\rangle) \\
(Next) & Sel(\langle e_1 \prec\!\!\!\leftarrow \mathtt{n} = e_2\rangle,\, \mathtt{m},\, e) & \stackrel{ev}{\rightarrow} \quad Sel(e_1,\, \mathtt{m},\, \lambda x.\, e\, \langle x \prec\!\!\!\leftarrow \mathtt{n} = e_2\rangle), \quad \mathtt{m} \neq \mathtt{n}
\end{array}
$$

Figure 1 – Reduction Semantics (Small-Step)

with object reclassification. In Section 8 we discuss the related work. The complete set of type assignment rules appears in the Appendix.

This manuscript cannot be considered a mere extended version of the paper [DGHL98]. In fact, with respect to that contribution, in the present work we have slightly changed the reduction semantics, substantially refined the type system, fully documented the proofs, and, in the last two novel sections, we have connected our approach with the most related developments in the area.

2 The Lambda Calculus of Objects

In this section, we present the Lambda Calculus of Objects $\lambda\mathcal{O}bj+$. The terms are defined by the following abstract grammar:

$$
\begin{array}{lll}
e \quad ::= & c \mid x \mid \lambda x.e \mid e_1\, e_2 \mid & (\lambda\text{-calculus}) \\
& \langle\,\rangle \mid \langle e_1 \prec\!\!\!\leftarrow \mathtt{m} = e_2\rangle \mid e \Leftarrow \mathtt{m} \mid & (\text{object terms}) \\
& Sel(e_1, \mathtt{m}, e_2) & (\text{auxiliary operation})
\end{array}
$$

where c is a meta-variable ranging over a set of constants, x is a meta-variable ranging over a set of variables, and $\mathtt{m}$ is a meta-variable ranging over a set of method names. As usual, terms that differ only in the names of bound variables are identified. The intended meaning of the object terms is the following: $\langle\,\rangle$ stands for the empty object; $\langle e_1 \prec\!\!\!\leftarrow \mathtt{m} = e_2\rangle$ stands for extending/overriding the object e_1 with a method $\mathtt{m}$ whose body is e_2; $e \Leftarrow \mathtt{m}$ stands for the result of sending the message $\mathtt{m}$ to the object e.

The auxiliary operation $Sel(e_1, \mathtt{m}, e_2)$ searches the body of the $\mathtt{m}$ method within the object e_1. In the recursive search of $\mathtt{m}$, $Sel(e_1, \mathtt{m}, e_2)$ removes methods from e_1; for this reason we need to introduce the expression e_2, which denotes a function that, applied to e_1, reconstructs the original object with the complete list of its methods. This function is peculiar to the operational semantics and, in practice, could be made not available to the programmer.

2.1 Operational Semantics

We define the evaluation of $\lambda\mathcal{O}bj+$ terms by means of the reduction rules in Figure 1 (Small-Step semantics). The evaluation relation $\stackrel{ev}{\twoheadrightarrow}$ is then taken to be the symmetric, reflexive, transitive and contextual closure of $\stackrel{ev}{\rightarrow}$.

In addition to the standard $(Beta)$ rule for lambda calculus, the main operation on objects is method invocation, whose reduction is defined by the $(Select)$ rule. Sending

a message $\mathtt{m}$ to an object e which contains a method $\mathtt{m}$ reduces to $Sel(e, \mathtt{m}, \lambda x.\, x)$, where the arguments of Sel have the following intuitive meanings:

- (1^{st}-arg) is a sub-object of the receiver (or recipient) of the message;
- (2^{nd}-arg) is the message we want to send to the receiver;
- (3^{rd}-arg) is a function that transforms the first argument in the receiver.

By looking at the last two rewriting rules, one may note that the Sel function "scans" the recipient of the message until it finds the definition of the method we want to use. When it finds such a method, it applies its body to the recipient of the message. Notice how the Sel function carries over, in its search, all the informations necessary to reconstruct the original receiver of the message.

Proposition 2.1 *The $\stackrel{ev}{\to}$ reduction is Church-Rosser.*

A quite simple technique to prove the Church-Rosser property for the λ-calculus has been proposed by Takahashi [Tak95]. The technique is based on parallel reduction and on Takahashi translation. It works as follows: first one defines a parallel reduction $\rightrightarrows$ on λ-terms, where several redexes can be reduced in parallel; then one shows that for any term M there is a terms $M*$, i.e. the Takahashi translation, obtained from M by reducing a maximum set of redexes in paralell. It follows almost immediately that the $\rightrightarrows$ reduction satisfies the triangular property, hence the diamond property, and therefore the calculus in confluent.

With respect to the λ-calculus, $\lambda\mathcal{O}bj+$ contains, besides the λ-rule, reduction rules for object terms; however, the latter do not interfere with the former, hence Takahashi technique can be straightforwardly applied to the $\lambda\mathcal{O}bj+$ calculus.

A deterministic reduction relation $\stackrel{dev}{\to}$ may be defined on $\lambda\mathcal{O}bj+$ by restricting the set of contexts used in the contextual closure of the reduction relation. In detail, we restrict the contextual closure to the set of contexts generated by the following grammar:

$$C[\,] = [\,] \;\mid\; C[\,]e \;\mid\; C[\,] \Leftarrow \mathtt{m} \;\mid\; Sel(C[\,], \mathtt{m}, e)$$

Notice that this choice of contexts enforces a "lazy" evaluation strategy over terms. The set of values, i.e. the terms where no reduction is possible, concides with the ones generated by the following grammar:

$$\begin{array}{lcl} obj & ::= & \langle\,\rangle \mid \langle e_1 \prec\!\!\!\!\leftarrow \mathtt{m} = e_2 \rangle \\ v & ::= & c \mid \lambda x.e \mid obj \end{array}$$

We conjecture that the deterministic reduction strategy $\stackrel{dev}{\to}$ is complete in the following sense.

Conjecture 2.2 (Completeness of $\stackrel{dev}{\to}$) *If $e \stackrel{ev}{\to} v$, then there exists v' such that $e \stackrel{dev}{\to} v'$.*

2.2 A Collection of Examples

Let $\langle \mathtt{m}_1 = e_1, \ldots, \mathtt{m}_k = e_k \rangle$ be syntactic sugar for $\langle \ldots \langle \langle\,\rangle \prec\!\!\!- \mathtt{m}_1 = e_1 \rangle \ldots \prec\!\!\!- \mathtt{m}_k = e_k \rangle$, for $\mathtt{k} \geq 1$. In the next examples we show three objects, performing, respectively:

- one self-inflicted extension;
- two (nested) self-inflicted extensions;
- a self-inflicted extension "on the fly".

Example 2.1 Consider the object `self_ext`*, defined as follows:*

$$\mathtt{self\_ext} \triangleq \langle \mathtt{add\_n} = \lambda \mathtt{self}.\langle \mathtt{self} \prec\!\!\!- \mathtt{n} = \lambda \mathtt{s}.1 \rangle \rangle.$$

If we send the message `add_n` *to* `self_ext`*, then we have the following computation:*

$$\begin{aligned} \mathtt{self\_ext} \Leftarrow \mathtt{add\_n} &\xrightarrow{ev} \mathit{Sel}(\mathtt{self\_ext}, \mathtt{add\_n}, \lambda x.x) \\ &\xrightarrow{ev} (\lambda \mathtt{self}.\langle \mathtt{self} \prec\!\!\!- \mathtt{n} = \lambda \mathtt{s}.1 \rangle)\, \mathtt{self\_ext} \\ &\xrightarrow{ev} \langle \mathtt{self\_ext} \prec\!\!\!- \mathtt{n} = \lambda \mathtt{s}.1 \rangle, \end{aligned}$$

i.e. *the method* `n` *has been added to* `self_ext`*. On the other hand, if we send the message* `add_n` *twice to* `self_ext`*, the method* `n` *is only overridden with the same body; hence, we get an object which is "operationally equivalent" to the previous one.*

Example 2.2 Consider the object `inner_ext`*, defined as follows:*

$$\mathtt{inner\_ext} \triangleq \langle \mathtt{add\_m\_n} = \lambda \mathtt{self}.\langle \mathtt{self} \prec\!\!\!- \mathtt{m} = \lambda \mathtt{s}.\langle \mathtt{s} \prec\!\!\!- \mathtt{n} = \lambda \mathtt{s}'.1 \rangle \rangle \rangle.$$

If we send the message `add_m_n` *to* `inner_ext`*, then we obtain:*

$$\mathtt{inner\_ext} \Leftarrow \mathtt{add\_m\_n} \xrightarrow{ev} \langle \mathtt{inner\_ext} \prec\!\!\!- \mathtt{m} = \lambda \mathtt{s}.\langle \mathtt{s} \prec\!\!\!- \mathtt{n} = \lambda \mathtt{s}'.1 \rangle \rangle,$$

i.e. *the method* `m` *has been added to* `inner_ext`*. On the other hand, if we send first the message* `add_m_n` *and then* `m` *to* `inner_ext`*, both methods* `m` *and* `n` *are added:*

$$\begin{aligned} &(\mathtt{inner\_ext} \Leftarrow \mathtt{add\_m\_n}) \Leftarrow \mathtt{m} \xrightarrow{ev} \\ &\langle \mathtt{add\_m\_n} = \lambda \mathtt{self}.\langle \mathtt{self} \prec\!\!\!- \mathtt{m} = \lambda \mathtt{s}.\langle \mathtt{s} \prec\!\!\!- \mathtt{n} = \lambda \mathtt{s}'.1 \rangle \rangle, \\ &\ \mathtt{m} = \lambda \mathtt{self}.\langle \mathtt{self} \prec\!\!\!- \mathtt{n} = \lambda \mathtt{s}'.1 \rangle, \\ &\ \mathtt{n} = \lambda \mathtt{self}.1 \rangle. \end{aligned}$$

Example 2.3 Consider the object `fly_ext`*, defined as follows:*

$$\begin{aligned} \mathtt{fly\_ext} \triangleq \langle &\mathtt{f} = \lambda \mathtt{self}.\lambda \mathtt{p}.\mathtt{p} \Leftarrow \mathtt{n}, \\ &\mathtt{get\_f} = \lambda \mathtt{self}.(\mathtt{self} \Leftarrow \mathtt{f}) \langle \mathtt{self} \prec\!\!\!- \mathtt{n} = \lambda \mathtt{s}.1 \rangle \rangle. \end{aligned}$$

If we send the message `get_f` *to* `fly_ext`*, then we get the following computation:*

$$\begin{aligned} &\mathtt{fly\_ext} \Leftarrow \mathtt{get\_f} \xrightarrow{ev} \mathit{Sel}(\mathtt{fly\_ext}, \mathtt{get\_f}, \lambda \mathtt{s}.\mathtt{s}) \\ &\xrightarrow{ev} (\lambda \mathtt{self}.(\mathtt{self} \Leftarrow \mathtt{f}) \langle \mathtt{self} \prec\!\!\!- \mathtt{n} = \lambda \mathtt{s}.1 \rangle)\, \mathtt{fly\_ext} \\ &\xrightarrow{ev} (\mathtt{fly\_ext} \Leftarrow \mathtt{f}) \langle \mathtt{fly\_ext} \prec\!\!\!- \mathtt{n} = \lambda \mathtt{s}.1 \rangle \\ &\xrightarrow{ev} \mathit{Sel}(\mathtt{fly\_ext}, \mathtt{f}, \lambda x.x)\, \langle \mathtt{fly\_ext} \prec\!\!\!- \mathtt{n} = \lambda \mathtt{s}.1 \rangle \\ &\xrightarrow{ev} (\lambda \mathtt{self}.\lambda \mathtt{p}.\mathtt{p} \Leftarrow \mathtt{n})\, \mathtt{fly\_ext}\, \langle \mathtt{fly\_ext} \prec\!\!\!- \mathtt{n} = \lambda \mathtt{s}.1 \rangle \\ &\xrightarrow{ev} \langle \mathtt{fly\_ext} \prec\!\!\!- \mathtt{n} = \lambda \mathtt{s}.1 \rangle \Leftarrow \mathtt{n} \\ &\xrightarrow{ev} 1, \end{aligned}$$

i.e. *the following steps are performed:*

1. *the method* `get_f` *calls the method* `f` *with actual parameter the object itself augmented with the* `n` *method;*
2. *the* `f` *method takes as input the host object augmented with the* `n` *method, and sends to this object the message* `n` *which simply returns the integer* `1`.

3 The Type System

In this section, we introduce the syntax of types, together with the most interesting type rules. In the sake of simplicity, we prefer to first present the type system without the rules related with object subsumption (which will be discussed in Section 4). The complete syntax and set of rules can be found in the Appendix.

3.1 Types

The type expressions are described by the following grammar:

$$\begin{array}{lcll} \tau & ::= & \iota \mid \tau \to \tau \mid \rho & \text{(generic types)} \\ \rho & ::= & t \mid \mathtt{pro}\, t.\langle\langle R\rangle\rangle \mid \rho \lhd \mathtt{m} & \text{(object-types)} \\ R & ::= & \varepsilon \mid R, \mathtt{m} : \sigma & \text{(rows)} \\ \kappa & ::= & T & \text{(kind of types)} \end{array}$$

In the rest of the article we will use τ and υ as meta-variables ranging over generic types, ι over constant types, ρ and ξ over object-types, and σ over types of methods. Moreover, t is a type variable, R a metavariable ranging over rows, $\mathtt{m}$ a method label, and κ a metavariable ranging over the unique kind of types T.

If M is a list of method labels ($M \equiv \mathtt{m}_1, \ldots, \mathtt{m}_n$), then we will use $\rho \lhd \langle M\rangle$ as syntactic sugar for $(\ldots((\rho \lhd \mathtt{m}_1) \lhd \mathtt{m}_2)\ldots \lhd \mathtt{m}_n)$. Object-types in the form $\mathtt{pro}\, t.\langle\langle R\rangle\rangle \lhd \langle M\rangle$ are called `pro`*-types*, where `pro` is a binder for the type-variable t and R is a *row*, i.e. an *unordered* set of pairs *(method label, method type)*. As in [FHM94], we may consider object-types as a form of recursively-defined types. In this paper we will freely use α-conversion of type-variables bound by `pro`.

The intended meaning of a `pro`-type:

$$\mathtt{pro}\, t.\langle\langle \mathtt{m}_1 : \sigma_1 \ldots \mathtt{m}_h : \sigma_h\rangle\rangle \lhd \mathtt{m}_{i_1} \lhd \ldots \lhd \mathtt{m}_{i_n}$$

is the following:

- $\mathtt{m}_1, \ldots, \mathtt{m}_h$ are the methods which are *present* in the `pro`-type;
- the methods among $\mathtt{m}_1, \ldots, \mathtt{m}_h$ that belong also to the list $\mathtt{m}_{i_1}, \ldots, \mathtt{m}_{i_n}$ are the methods that are *available* and can be invoked. Therefore, the object-type $\mathtt{pro}\, t.\langle\langle \mathtt{m}_1 : \sigma_1 \ldots \mathtt{m}_h : \sigma_h\rangle\rangle \lhd \langle \mathtt{m}_1, \ldots, \mathtt{m}_h\rangle$ is the counterpart of the object-type $\mathtt{class}\, t.\langle\langle \mathtt{m}_1 : \sigma_1 \ldots \mathtt{m}_h : \sigma_h\rangle\rangle$ of [FHM94];
- the methods in $\mathtt{m}_1, \ldots, \mathtt{m}_h$ that do not appear in the list $\mathtt{m}_{i_1}, \ldots, \mathtt{m}_{i_n}$ are methods that cannot be invoked: they are just *reserved*. We can extend an object e with a new method $\mathtt{m}$ having type σ only if it is possible to assign to e an object-type of the form $\mathtt{pro}\, t.\langle\langle R, \mathtt{m} : \sigma\rangle\rangle \lhd \langle M\rangle$. As remarked in the introduction, this reservation mechanism is crucial to guarantee the consistency of the type system.

The operator "$\lhd$" is used to make active and usable those methods that were previously just reserved in an object-type; essentially, $\lhd$ is the "type counterpart" of the operator $\prec\!\!-$.

In the sequel we will use the following notations, for R a row and M a list:

$$\begin{array}{lcl} \mathcal{M}(R) & \triangleq & \{\mathtt{m} \mid (\mathtt{m} : \sigma) \in R\} \\ \{M\} & \triangleq & \text{the set of methods contained in M} \\ M - \mathtt{m} & \triangleq & \text{M without the occurrence of the } \mathtt{m} \text{ method} \end{array}$$

3.2 Contexts and Judgments

The contexts have the following form:

$$\Gamma ::= \varepsilon \mid \Gamma, x : \tau \mid \Gamma, t <\!\!\# \rho$$

Our type assignment system uses judgments of the following shapes:

$$\Gamma \vdash ok \qquad \Gamma \vdash \tau : T \qquad \Gamma \vdash e : \tau \qquad \Gamma \vdash \xi <\!\!\# \rho$$

The intended meaning of the first three judgments is standard: well-formed contexts and types, and assignment of type τ to term e. The intended meaning of $\Gamma \vdash \xi <\!\!\# \rho$ is that ξ is the type of a possible extension of an object having type ρ. As in [Bru94, BB99], this judgment formalizes the notion of *method-specialization* (or *protocol-extension*), *i.e.* the capability to "inherit" the type of the methods of the prototype.

The type rules for well-formed contexts are quite standard. We just comment that in the rule $(Cont{-}t)$ we impose that the object-types used to bind variables are not variable types themselves: this condition does not have any serious restriction, and has been set in the type system in order to make simpler the proofs of its properties.

3.3 Well-formed Types Rules

The $(Type{-}Pro)$ rule

$$\frac{\Gamma, t <\!\!\# \mathtt{pro}\, t.\langle\!\langle R \rangle\!\rangle \vdash \sigma : T \qquad \mathtt{m} \notin \mathcal{M}(R)}{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma \rangle\!\rangle : T}$$

asserts that the object-type $\mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma \rangle\!\rangle$ is well-formed if the object-type $\mathtt{pro}\, t.\langle\!\langle R \rangle\!\rangle$ is well-formed and the type σ is well-formed under the hypothesis that t is an object-type containing the methods in $\mathcal{M}(R)$. Since σ may contain a subexpression in the form $t \lhd \mathtt{n}$, with $\mathtt{n} \in \mathcal{M}(R)$, we need to introduce in the context the hypothesis $t <\!\!\# \mathtt{pro}\, t.\langle\!\langle R \rangle\!\rangle$ to prove that $t \lhd \mathtt{n}$ is a well-formed type.

The $(Type{-}Extend)$ rule

$$\frac{\Gamma \vdash \rho <\!\!\# \mathtt{pro}\, t.\langle\!\langle R \rangle\!\rangle \qquad \{M\} \subseteq \mathcal{M}(R)}{\Gamma \vdash \rho \lhd \langle M \rangle : T}$$

asserts that in order to activate the methods M in the object type ρ, the methods M need to be present (reserved) in ρ.

3.4 Matching Type Rules

The $(Match{-}Pro)$ rule

$$\frac{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R' \rangle\!\rangle \triangleleft \langle M' \rangle : T \qquad \Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R \rangle\!\rangle \triangleleft \langle M \rangle : T \qquad R \subseteq R' \qquad \{M\} \subseteq \{M'\}}{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R' \rangle\!\rangle \triangleleft \langle M' \rangle <\!\# \mathtt{pro}\, t.\langle\!\langle R \rangle\!\rangle \triangleleft \langle M \rangle}$$

asserts that an object-type with more reserved and more available methods specializes an object-type with less reserved and less available methods.

The $(Match{-}Var)$ rule

$$\frac{\Gamma, t <\!\# \rho, \Gamma' \vdash \rho \triangleleft \langle M \rangle <\!\# \xi}{\Gamma, t <\!\# \rho, \Gamma' \vdash t \triangleleft \langle M \rangle <\!\# \xi}$$

makes available the matching judgments present in the context. It asserts that, if a context contains the hypothesis that a type variable t specializes a type ρ, and ρ itself, incremented with a set of methods M, specializes a type ξ, then, by transitivity of the matching relation, t, incremented by the methods in M, specializes ξ.

The $(Match{-}t)$ rule

$$\frac{\Gamma \vdash t \triangleleft \langle M' \rangle : T \qquad \{M\} \subseteq \{M'\}}{\Gamma \vdash t \triangleleft \langle M' \rangle <\!\# t \triangleleft \langle M \rangle}$$

concerns object-types built from the same type variable, simply asserting that an object-type with more available methods specializes an object-type with less available methods.

3.5 Type Rules for Terms

The set of type rules for lambda terms are self-explanatory and hence they need no further justification. The $(Empty)$ rule assigns to an empty object an empty `pro`-type.

The $(Pre{-}Extend)$ rule

$$\frac{\Gamma \vdash e : \mathtt{pro}\, t.\langle\!\langle R \rangle\!\rangle \triangleleft \langle M \rangle \qquad \Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R, R' \rangle\!\rangle \triangleleft \langle M \rangle : T}{\Gamma \vdash e : \mathtt{pro}\, t.\langle\!\langle R, R' \rangle\!\rangle \triangleleft \langle M \rangle}$$

asserts that an object e having type $\mathtt{pro}\, t.\langle\!\langle R \rangle\!\rangle \triangleleft \langle M \rangle$ can be considered also an object having type $\mathtt{pro}\, t.\langle\!\langle R, R' \rangle\!\rangle \triangleleft \langle M \rangle$, i.e. with more reserved methods. This rule has to be used in conjunction with the $(Extend)$ one; it ensures that we can dynamically add fresh methods. Notice that $(Pre{-}Extend)$ cannot be applied when e is the variable `self`; in fact, as explained in the Remark 3.1 below, the type of `self` can only be a type variable. This fact is crucial for the soundness of the type system.

The $(Extend)$ rule

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \rho \\ \Gamma \vdash \rho <\!\# \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma \rangle\!\rangle \triangleleft \langle M \rangle \\ \Gamma, t <\!\# \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma \rangle\!\rangle \triangleleft \langle M, \mathtt{m} \rangle \vdash e_2 : t \to \sigma\end{array}}{\Gamma \vdash \langle e_1 \prec\!\!\!- \mathtt{m} = e_2 \rangle : \rho \triangleleft \mathtt{m}}$$

can be applied in the following cases:

1. when the object e_1 has type $\mathtt{pro}\, t.\langle\langle R, \mathtt{m} : \sigma\rangle\rangle \triangleleft \langle M\rangle$ (or, by a previous application of the *(Pre-Extend)* rule, $\mathtt{pro}\, t.\langle\langle R\rangle\rangle \triangleleft \langle M\rangle$). In this case the object e_1 is *extended* with the (fresh) method $\mathtt{m}$;

2. when ρ is a type variable t. In this case e_1 can be the variable $\mathtt{self}$ and there is a *self-inflicted extension*.

The bound for t is the same as the final type for the object $\langle e_1 \prec\!\!\!- \mathtt{m} = e_2\rangle$; this allows a recursive call of the method $\mathtt{m}$ inside the expression e_2, defining the method $\mathtt{m}$ itself.

The (*Override*) rule

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \rho \\ \Gamma \vdash \rho <\!\!\# \mathtt{pro}\, t.\langle\langle R, \mathtt{m} : \sigma\rangle\rangle \triangleleft \langle M, \mathtt{m}\rangle \\ \Gamma, t <\!\!\# \mathtt{pro}\, t.\langle\langle R, \mathtt{m} : \sigma\rangle\rangle \triangleleft \langle M, \mathtt{m}\rangle \vdash e_2 : t \to \sigma\end{array}}{\Gamma \vdash \langle e_1 \prec\!\!\!- \mathtt{m} = e_2\rangle : \rho}$$

is quite similar to the (*Extend*) rule, but it is applied when the method $\mathtt{m}$ is *already* available in the object e_1, hence the body of $\mathtt{m}$ is *overridden* with a new body.

Remark 3.1 *By inspecting the* (*Extend*) *and* (*Override*) *rules, one can see why the type of* $\mathtt{self}$ *is always a type variable. In fact, the body* e_2 *of the new added method needs to have type* $t \to \sigma$*. Therefore, if* e_2 *reduces to a value, this value has to be a lambda abstraction in the form* $\lambda \mathtt{self}.e_2'$*. It follows that, in assigning a type to* e_2'*, we must use a context containing the hypothesis* $\mathtt{self} : t$*. Since no subsumption rule is available, the only type we can deduce for* $\mathtt{self}$ *is* t*.* □

The (*Send*) rule

$$\frac{\Gamma \vdash e : \rho \qquad \Gamma \vdash \rho <\!\!\# \mathtt{pro}\, t.\langle\langle R, \mathtt{m} : \sigma\rangle\rangle \triangleleft \langle M, \mathtt{m}\rangle}{\Gamma \vdash e \Leftarrow \mathtt{m} : \sigma[\rho/t]}$$

is the standard rule that one can expect in a type system based on matching. We require that the method we are invoking is available in the recipient of the message.

The (*Select*) is the following

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \rho \\ \Gamma \vdash \rho <\!\!\# \mathtt{pro}\, t.\langle\langle R, \mathtt{m} : \sigma\rangle\rangle \triangleleft \langle M, \mathtt{m}\rangle \\ \Gamma, t <\!\!\# \mathtt{pro}\, t.\langle\langle R, \mathtt{m} : \sigma\rangle\rangle \triangleleft \langle M, \mathtt{m}\rangle \vdash e_2 : t \to (t \triangleleft N)\end{array}}{\Gamma \vdash Sel(e_1, \mathtt{m}, e_2) : \sigma[(\rho \triangleleft N)/t]}$$

The first two conditions ensure that the $\mathtt{m}$ method is available in e_1, while the last condition that e_2 is a function that transforms an object into a more refined one.

4 Adding Object Subsumption

In this section, we propose an extension of the type assignment system for $\lambda\mathcal{O}bj+$, introduced in Section 3, to accommodate width-subtyping.

It is well-known that adding a subsumption rule over terms, that is:

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash \tau <\!\!\# \upsilon}{\Gamma \vdash e : \upsilon}$$

clearly increases the set of expressions which are typable in the type system. Unfortunately, this rule is *unsound* in the presence of object extension; in fact, we could (by subsumption) first hide a method in an object, and then add it again with a type incompatible with the previous one (see [AC96, FM94]).

Therefore, in order to include subsumption rules in our type assignment system, we need to introduce another "kind" of object-types, namely:

$$\texttt{obj}\, t.\langle\!\langle R \rangle\!\rangle \triangleleft \langle M \rangle$$

The main difference between the `pro`-types and the `obj`-types consists in the fact that the *(Pre-Extend)* rule (see Section 3.5) cannot be applied when an object has type $\texttt{obj}\, t.\langle\!\langle R \rangle\!\rangle \triangleleft \langle M \rangle$; it follows that the type $\texttt{obj}\, t.\langle\!\langle R \rangle\!\rangle \triangleleft \langle M \rangle$ permits extensions of an object only by enriching the list M, i.e. by making active its reserved methods. This approach to subsumption is reminiscent of the one in [FM95, Liq97].

We also define a sub-kind of the kind T of types, termed Rgd, whose intended meaning is the subset of the *rigid*, *i.e. non-extensible* types.

In the end, the new syntax of types and kinds is:

$$\begin{array}{rcl} \rho & ::= & \ldots \mid \texttt{obj}\, t.\langle\!\langle R \rangle\!\rangle \\ \kappa & ::= & T \mid Rgd \end{array}$$

The subset of rigid types contains the `obj`-types and is closed under the arrow constructor. In order to axiomatize this, we introduce a new form of judgment:

$$\Gamma \vdash \tau : Rgd$$

which we will freely use to deal also with rows, by writing $\Gamma \vdash R : Rgd$. The typing rules that define the new judgment are presented in Appendix B.

In fact, the subsumption rule is valid only when the type in the conclusion is rigid:

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash \tau \lessdot\!\!\# \upsilon \qquad \Gamma \vdash \upsilon : Rgd}{\Gamma \vdash e : \upsilon}\ (Subsume)$$

It is important to point out that, so doing, we do *not* need to introduce another partial order on types, *i.e.* an ordinary subtyping relation, to deal with subsumption. By introducing the sub-kind of rigid types, we make the matching relation compatible with subsumption, and hence we can make it play the role of the width-subtyping relation. This is in sharp contrast with the uses of matching proposed in the literature ([Bru94, BPF97, BB99]). Hence, we can suggestively say that in our type assignment system *"matching is a relation on types compatible with a limited subsumption rule"*.

4.1 Extra Rules for Subsumption

Extra rules for the `obj`-types are necessary. Some of these rules are simply a rephrasing of the rules presented so far, replacing the binder `pro` by the binder `obj`. The most important new rules are $(Rgd{-}Obj)$ and $(Promote)$.

The $(Rgd{-}Obj)$ rule

$$\frac{\Gamma \vdash \texttt{obj}\, t.\langle\!\langle R \rangle\!\rangle \triangleleft \langle M \rangle : T \qquad t \text{ covariant in } R \qquad \Gamma \vdash R : Rgd}{\Gamma \vdash \texttt{obj}\, t.\langle\!\langle R \rangle\!\rangle \triangleleft \langle M \rangle : Rgd}$$

asserts that subsumption is unsound for methods having t in contravariant position with respect to the arrow type constructor. Therefore, the variable t is forced to

occur only covariantly in the methods of R. Hence, *binary methods* are lost. This is, unfortunately, a common price to pay in order to have a fully static type system with subtyping (see [BCC+96, Cas95, Cas96]).

The (*Promote*) rule

$$\frac{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R' \rangle\!\rangle \triangleleft \langle M' \rangle : T \qquad \Gamma \vdash \mathtt{obj}\, t.\langle\!\langle R \rangle\!\rangle \triangleleft \langle M \rangle : T \qquad R \subseteq R' \qquad \{M\} \subseteq \{M'\}}{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R' \rangle\!\rangle \triangleleft \langle M' \rangle <\!\# \mathtt{obj}\, t.\langle\!\langle R \rangle\!\rangle \triangleleft \langle M \rangle}$$

used together with (*Subsume*), "promotes" a fully-specializable `pro`-type into a limitedly specializable `obj`-type with less reserved and less available methods.

5 A Collection of Examples

In this section, we give the types of the examples presented in Section 2.2, together with some other (hopefully) motivating examples.

The objects `self_ext`, `inner_ext`, and `fly_ext`, of Examples 2.1, 2.2, and 2.3, respectively, can be given the following types:

$$\mathtt{self\_ext} : \mathtt{pro}\, t.\langle\!\langle \mathtt{add\_n} : t \triangleleft \mathtt{n}, \mathtt{n} : \mathtt{int} \rangle\!\rangle \triangleleft \mathtt{add\_n}$$

$$\mathtt{inner\_ext} : \mathtt{pro}\, t.\langle\!\langle \mathtt{add\_m\_n} : t \triangleleft \mathtt{m}, \mathtt{m} : t \triangleleft \mathtt{n}, \mathtt{n} : \mathtt{int} \rangle\!\rangle \triangleleft \mathtt{add\_m\_n}$$

$$\mathtt{fly\_ext} : \mathtt{pro}\, t.\langle\!\langle \mathtt{f} : t \triangleleft \mathtt{n}, \mathtt{get\_f} : (t \triangleleft \mathtt{n}) \to \mathtt{int}, \mathtt{n} : \mathtt{int} \rangle\!\rangle \triangleleft \langle \mathtt{f}, \mathtt{get\_f} \rangle.$$

A possible derivation for `self_ext` is presented in Figure 2.

Example 5.1 We show how class declaration can be simulated in $\lambda\mathcal{O}bj+$ and how using the self-inflicted extension we can factorize in a single declaration the definition of a hierarchy of classes. Let the method `add_set_col` *be defined as in Example 1.2, and let us consider the simple class definition of* `PointClass`*:*

$$\mathtt{PointClass} \triangleq \langle \mathtt{new} = \lambda \mathtt{s}.\langle \mathtt{x} = \lambda \mathtt{self}.1, \mathtt{add\_set\_col} = \ldots \rangle \rangle.$$

The object `PointClass` *can be used to create instances of both points and colored points, by using the expressions* `PointClass` $\Leftarrow$ `new` *and* (`PointClass` $\Leftarrow$ `new`) $\Leftarrow$ `add_set_col`*.*

$$\dfrac{\dfrac{\dfrac{\Gamma \vdash \mathtt{s} : t \qquad \Gamma \vdash t <\!\# \mathtt{pro}\, t.\langle\!\langle \mathtt{n} : \mathtt{int} \rangle\!\rangle \qquad \Gamma, t' <\!\# \mathtt{pro}\, t'.\ \langle\!\langle \mathtt{n} : \mathtt{int} \rangle\!\rangle \triangleleft \mathtt{n} \vdash \lambda \mathtt{s}'.1 : t' \to \mathtt{int}}{\Gamma \vdash \langle \mathtt{s} \prec\!\!\!- \mathtt{n} = \lambda \mathtt{s}'.1 \rangle : t \triangleleft \mathtt{n}}\ (Extend)}{t <\!\# \rho \triangleleft \mathtt{add\_n} \vdash \lambda \mathtt{s}.\langle \mathtt{s} \prec\!\!\!- \mathtt{n} = \lambda \mathtt{s}'.1 \rangle : (t \to (t \triangleleft \mathtt{n}))}\ (Abs) \qquad \dfrac{\varepsilon \vdash \langle\,\rangle : \rho^{(*)} \qquad \varepsilon \vdash \rho <\!\# \rho}{\varepsilon \vdash \langle \mathtt{add\_n} = \lambda \mathtt{s}.\langle \mathtt{s} \prec\!\!\!- \mathtt{n} = \lambda \mathtt{s}'.1 \rangle \rangle : \rho \triangleleft \mathtt{add\_n}}\ (Extend)$$

where $\rho \equiv \mathtt{pro}\, t.\langle\!\langle \mathtt{add\_n} : t \triangleleft \mathtt{n}, \mathtt{n} : \mathtt{int} \rangle\!\rangle$, $\Gamma \equiv t <\!\# \rho \triangleleft \mathtt{add\_n},\ \mathtt{s} : t$, and the judgment $(*)$ is derivable by the $(Pre{-}Extend)$ rule.

Figure 2 – A derivation for `self_ext`

Example 5.2 (Subsumption 1) We show how subsumption can interact with object extension. Let be:

$$\begin{array}{lcl} \mathtt{P}' & \triangleq & \mathtt{obj}\, t.\langle\langle \mathtt{x} : \mathtt{int}, \mathtt{col} : \mathtt{bool}\rangle\rangle \lhd \mathtt{x} \\ \mathtt{CP} & \triangleq & \mathtt{obj}\, t.\langle\langle \mathtt{x} : \mathtt{int}, \mathtt{col} : \mathtt{bool}\rangle\rangle \lhd \langle \mathtt{x}, \mathtt{col}\rangle \\ \mathtt{g} & \triangleq & \lambda p.\langle p \prec\!\!\!\leftarrow \mathtt{col} = \lambda \mathtt{s}.\mathtt{true}\rangle, \end{array}$$

and let `point` *and* `col_point` *be of type* $\mathtt{P}'$ *and* `CP`*, respectively.*

By the type assignment rules we have:

$$\varepsilon \vdash \mathtt{CP} <\!\!\!\# \mathtt{P}'$$

$$\varepsilon \vdash \mathtt{g} : \mathtt{P}' \to \mathtt{CP}$$

$$\varepsilon \vdash \mathtt{g}\,(\mathtt{col\_point}) : \mathtt{CP}$$

$$\varepsilon \vdash (\lambda \mathtt{f}.\mathtt{f}(\mathtt{point}) \Leftarrow \mathtt{col} == \mathtt{f}(\mathtt{col\_point}) \Leftarrow \mathtt{col})\,\mathtt{g} : \mathtt{bool},$$

where the equality function $==:\ t \to t \to \mathtt{bool}$ *is used in infix notation. Notice that the object:*

$$(\lambda \mathtt{f}.\mathtt{f}(\mathtt{point}) \Leftarrow \mathtt{col} == \mathtt{f}(\mathtt{col\_point}) \Leftarrow \mathtt{col})$$

would not be typable without the subsumption rule.

Example 5.3 (Subsumption 2) We show how self-inflicted extension can interact with object subsumption. Let be:

$$\begin{array}{lcl} \mathtt{P} & \triangleq & \mathtt{obj}\, t.\langle\langle \mathtt{x} : \mathtt{int}\rangle\rangle \lhd \mathtt{x} \\ \mathtt{o} & \triangleq & \langle \mathtt{copy\_x} = \lambda \mathtt{self}.\lambda \mathtt{p}.\langle \mathtt{self} \prec\!\!\!\leftarrow \mathtt{x} = \lambda \mathtt{s}.\mathtt{p} \Leftarrow \mathtt{x}\rangle\rangle. \end{array}$$

By the type assignment rules we have:

$$\varepsilon \vdash \mathtt{o} : \mathtt{pro}\, t.\langle\langle \mathtt{copy\_x} : \mathtt{P} \to (\mathtt{t} \lhd \mathtt{x}), \mathtt{x} : \mathtt{int}\rangle\rangle \lhd \mathtt{copy\_x}$$

$$\varepsilon \vdash \mathtt{o} \Leftarrow \mathtt{copy\_x}\,(\mathtt{col\_point}) : \xi$$

$$\varepsilon \vdash \mathtt{o} \Leftarrow \mathtt{copy\_x}\,(\mathtt{col\_point}) \Leftarrow \mathtt{copy\_x}\,(\mathtt{point}) : \xi,$$

where $\xi \triangleq \mathtt{pro}\, t.\langle\langle \mathtt{x} : \mathtt{int}, \mathtt{copy\_x} : \mathtt{P} \to \mathtt{t}\rangle\rangle \lhd \langle \mathtt{x}, \mathtt{copy\_x}\rangle$.

Notice that the object $\mathtt{o} \Leftarrow \mathtt{copy\_x}\,(\mathtt{col\_point}) \Leftarrow \mathtt{copy\_x}\,(\mathtt{point})$ *would not be typable without the subsumption rule.*

Example 5.4 (Downcasting) The self-inflicted extension permits to perform explicit downcasting simply by method calling. In fact, let `point` *and* `col_point` *be objects with* `equal` *methods (checking the values of* `x`*, and* `x`, `col`*, respectively) and* `add_set_col` *the self-extension method presented in Example 1.2, typable as follows:*

$$\varepsilon \vdash \mathtt{point} : \mathtt{pro}\, t.R \qquad \textit{and} \qquad \varepsilon \vdash \mathtt{col\_point} : \mathtt{pro}\, t.R \lhd \mathtt{col},$$

where R *is:*

$$\langle\langle \mathtt{x} : \mathtt{int}, \mathtt{equal} : t \to \mathtt{bool}, \mathtt{add\_set\_col} : t \lhd \mathtt{col}, \mathtt{col} : \mathtt{bool}\rangle\rangle \lhd \langle \mathtt{x}, \mathtt{add\_set\_col}\rangle.$$

Then, the following judgments are derivable:

$$\begin{array}{lcl} \varepsilon & \vdash & \mathtt{col\_point} \Leftarrow \mathtt{equal} : \mathtt{pro}\, t.R \lhd \mathtt{col} \to \mathtt{bool} \\ \varepsilon & \vdash & \mathtt{point} \Leftarrow \mathtt{add\_set\_col} : \mathtt{pro}\, t.R \lhd \mathtt{col} \\ \varepsilon & \vdash & \mathtt{col\_point} \Leftarrow \mathtt{equal}(\mathtt{point} \Leftarrow \mathtt{add\_set\_col}) : \mathtt{bool}. \end{array}$$

6 Soundness of the Type System

In this section, we prove the crucial property of our type system, *i.e.* Theorem 6.9, the Subject Reduction Theorem. It needs a preliminary series of technical lemmas presenting basic and technical properties, which are proved by complex, albeit standard, inductive arguments. As a corollary of the Theorem 6.9, we shall derive the fundamental result of the paper, *i.e.* the Type Soundness of our typing discipline.

We will first consider the *plain* type assignment system without subsumption, then we will extend the Subject Reduction Theorem to the whole type system.

In the presentation of the properties and theorems, we will use A as metavariable ranging on the statements in the form ok, $\tau : T$, $\rho <\!\!\# \xi$ and $e : \tau$; we will use C as metavariable for statements in the form $x : \tau$ and $t <\!\!\# \rho$.

Lemma 6.1 *(Sub-derivation)*

(i) If Δ is a derivation of $\Gamma, \Gamma' \vdash A$, then there exists a sub-derivation $\Delta' \subseteq \Delta$ of $\Gamma \vdash ok$.

(ii) If Δ is a derivation of $\Gamma, x : \sigma, \Gamma' \vdash A$, then there exists a sub-derivation $\Delta' \subseteq \Delta$ of $\Gamma \vdash \sigma : T$.

(iii) If Δ is a derivation of $\Gamma, t <\!\!\# \sigma, \Gamma' \vdash A$, then there exists a sub-derivation $\Delta' \subseteq \Delta$ of $\Gamma \vdash \sigma : T$.

The three points are proved, separately, by structural induction on the derivation Δ.

(i) The only cases where the inductive hypothesis cannot be applied are the cases where the last rule in Δ is a context rule, that is, the only kind of rule that can increase the context, and Γ' is empty. In these cases the thesis coincides with the hypothesis. For all the other cases the thesis follows immediately by an application of the inductive hypothesis.

(ii) As in point (i), either we conclude immediately by inductive hypothesis or we are in the case where Γ' is empty and the last rule in Δ is a context rule. In this latter case the last rule in Δ is necessarily a rule $(Cont{-}x)$ deriving $\Gamma, x : \sigma \vdash ok$, and the first premise of this rule coincides with the thesis.

(iii) The proof works similarly to point (ii). □

Lemma 6.2 *(Weakening)*

(i) If $\Gamma, \Gamma' \vdash A$ and $\Gamma, C, \Gamma' \vdash ok$, then $\Gamma, C, \Gamma' \vdash A$.

(ii) If $\Gamma \vdash A$ and $\Gamma, \Gamma' \vdash ok$, then $\Gamma, \Gamma' \vdash A$.

(i) By structural induction on the derivation Δ of $\Gamma, \Gamma' \vdash A$. If the last rule in Δ has the context in the conclusion identical to the context of the premise(s), then it is possible to apply the inductive hypothesis, thus deriving almost immediately the goal. For the other cases, if the last rule in Δ is a rule $(Cont{-}x)$ or $(Cont{-}t)$, then the proof is trivial, since the second hypothesis coincides with the thesis. The remaining cases concern the rules $(Type{-}Pro)$, (Abs), $(Extend)$ and $(Override)$. These cases require a more careful treatment. We detail here only the proof for $(Type{-}Pro)$, since the other cases are handled in a similar way.

In the $(Type{-}Pro)$ case, the hypothesis $\Gamma, \Gamma' \vdash \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma \rangle\!\rangle : T$ follows from the judgment:

$$\Gamma, \Gamma', t <\!\!\# \mathtt{pro}\, t.\langle\!\langle R \rangle\!\rangle \vdash \sigma : T \tag{1}$$

Let us briefly remark that if the statement C of the second hypothesis is equal to $t \leqslant\!\!\# \rho$, for some type ρ, then it is convenient to α-convert the type $\mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma \rangle\!\rangle$ to avoid clash of variables. In any case, by Lemma 6.1.(iii) (Sub-derivation), there exists a sub-derivation of Δ deriving $\Gamma, \Gamma' \vdash \mathtt{pro}\, t.\langle\!\langle R \rangle\!\rangle : T$, from which, by inductive hypothesis, $\Gamma, C, \Gamma' \vdash \mathtt{pro}\, t.\langle\!\langle R \rangle\!\rangle : T$, and in turn, by the rule $(Cont{-}t)$, $\Gamma, C, \Gamma', t \leqslant\!\!\# \mathtt{pro}\, t.\langle\!\langle R \rangle\!\rangle \vdash ok$. By using (1) and the inductive hypothesis, we deduce $\Gamma, C, \Gamma', t \leqslant\!\!\# \mathtt{pro}\, t.\langle\!\langle R \rangle\!\rangle \vdash \sigma : T$. Finally we have the thesis via the rule $(Type{-}Pro)$.

(ii) By induction on the length of Γ'; the proof uses the previous point (i) and Lemma 6.1.(i) (Sub-derivation). □

Lemma 6.3 *(Well-formed types)*

(i) $\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R \rangle\!\rangle \lhd \langle M \rangle : T$ *if and only if* $\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R \rangle\!\rangle : T$ *and* $\{M\} \subseteq \mathcal{M}(R)$.

(ii) $\Gamma \vdash t \lhd \langle M \rangle : T$ *if and only if* Γ *contains* $t \leqslant\!\!\# \mathtt{pro}\, t.\langle\!\langle R \rangle\!\rangle \lhd \langle N \rangle$, *with* $\{M\} \subseteq \mathcal{M}(R)$.

Point (i) is immediately proved by inspection on the rules for well-formed types and matching. Point (ii) can be proved by inspection on the rules for well-formed context, well-formed types and matching. □

Notice that in the following proofs we often do not refer explicitly to the previous lemmas, considering obvious their application.

Proposition 6.4 *(Matching has well-formed types)*
If $\Gamma \vdash \xi \leqslant\!\!\# \rho$, *then* $\Gamma \vdash \xi : T$ *and* $\Gamma \vdash \rho : T$.

By structural induction on the derivation Δ of $\Gamma \vdash \xi \leqslant\!\!\# \rho$. The premises of the rule $(Match{-}Pro)$ are exactly the thesis. If the last rule in Δ is a $(Match{-}t)$ rule, we can conclude using the premise of the rule and Lemma 6.3.(ii) (Well-formed types). If the last rule applied in Δ is a $(Match{-}Var)$ rule , then the judgment $\Gamma, t \leqslant\!\!\# \xi, \Gamma' \vdash t \lhd \langle M \rangle \leqslant\!\!\# \rho$ is derived from the judgment $\Gamma, t \leqslant\!\!\# \xi, \Gamma' \vdash \xi \lhd \langle M \rangle \leqslant\!\!\# \rho$. By inductive hypothesis, ρ is well-typed, and $\Gamma, t \leqslant\!\!\# \xi, \Gamma' \vdash \xi \lhd \langle M \rangle : T$. By the side condition in $(Cont{-}t)$ rule, ξ must be in the form $\mathtt{pro}\, t.\langle\!\langle R \rangle\!\rangle \lhd \langle M' \rangle$, and, by Lemma 6.3.(i) (Well-formed types), $\{M\} \subseteq \mathcal{M}(R)$. We can now conclude $\Gamma, t \leqslant\!\!\# \xi, \Gamma' \vdash t \lhd \langle M \rangle : T$ via the $(Type{-}Extend)$ rule. □

Lemma 6.5 *(Matching)*

(i) $\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R \rangle\!\rangle \lhd \langle M \rangle \leqslant\!\!\# \rho$ *if and only if* $\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R \rangle\!\rangle \lhd \langle M \rangle : T$ *and* $\Gamma \vdash \rho : T$ *and* $\rho \equiv \mathtt{pro}\, t.\langle\!\langle R' \rangle\!\rangle \lhd \langle M' \rangle$, *with* $R' \subseteq R$ *and* $\{M'\} \subseteq \{M\}$.

(ii) $\Gamma \vdash \rho \leqslant\!\!\# t \lhd \langle M \rangle$ *if and only if* $\Gamma \vdash \rho : T$ *and* $\rho \equiv t \lhd \langle M' \rangle$, *with* $\{M\} \subseteq \{M'\}$.

(iii) $\Gamma \vdash t \lhd \langle M \rangle \leqslant\!\!\# \mathtt{pro}\, t.\langle\!\langle R \rangle\!\rangle \lhd \langle N \rangle$ *if and only if* Γ *contains* $t \leqslant\!\!\# \mathtt{pro}\, t.\langle\!\langle R' \rangle\!\rangle \lhd \langle N' \rangle$, *with* $R \subset R'$ *and* $\{N\} \subset \{M \cup N'\}$.

(iv) (Reflexivity) If $\Gamma \vdash \rho : T$ *then* $\Gamma \vdash \rho \leqslant\!\!\# \rho$.

(v) (Transitivity) If $\Gamma \vdash \rho_1 \leqslant\!\!\# \rho_2$ *and* $\Gamma \vdash \rho_2 \leqslant\!\!\# \rho_3$, *then* $\Gamma \vdash \rho_1 \leqslant\!\!\# \rho_3$.

(vi) (Uniqueness) If $\Gamma \vdash \rho \leqslant\!\!\# \mathtt{pro}\, t.\langle\!\langle R, m : \sigma \rangle\!\rangle$ *and* $\Gamma \vdash \rho \leqslant\!\!\# \mathtt{pro}\, t.\langle\!\langle R', m : \sigma' \rangle\!\rangle$, *then* $\sigma \equiv \sigma'$.

(vii) If $\Gamma \vdash \rho \leqslant\!\!\# \xi$ *and* $\Gamma \vdash \xi \lhd \mathtt{m} : T$, *then* $\Gamma \vdash \rho \lhd \mathtt{m} \leqslant\!\!\# \xi \lhd \mathtt{m}$.

(viii) *If* $\Gamma \vdash \rho \lhd \mathtt{m} <\!\!\# \, \mathtt{pro}\, t.\langle\!\langle R \rangle\!\rangle \lhd \langle M \rangle$, *then* $\Gamma \vdash \rho <\!\!\# \, \mathtt{pro}\, t.\langle\!\langle R \rangle\!\rangle \lhd \langle M - \mathtt{m} \rangle$.

(ix) *If* $\Gamma \vdash \rho \lhd \mathtt{m} : T$, *then* $\Gamma \vdash \rho \lhd \mathtt{m} <\!\!\# \, \rho$.

(i) (ii) (iii) The thesis is immediate by inspection on the matching rules.

(iv) By cases on the form of ρ. The thesis can be derived immediately using either the $(Match{-}Pro)$ rule or the $(Match{-}t)$ rule.

(v) By cases on the form of ρ_1, ρ_2, ρ_3, using the points (i), (ii), (iii) above. If $\rho_1 \equiv \mathtt{pro}\, t.\langle\!\langle R \rangle\!\rangle \lhd \langle M \rangle$, we conclude by a triple application of point (i). If $\rho_3 \equiv t \lhd \langle M \rangle$, we conclude by three applications of point (ii). If $\rho_1 \equiv t \lhd \langle M \rangle$ and $\rho_3 \equiv \mathtt{pro}\, t.\langle\!\langle R \rangle\!\rangle \lhd \langle M \rangle$, we conclude by reasoning on the form of ρ_2, using points (i), (ii), (iii).

(vi) By cases on the form of ρ, using either point (i) or point (iii).

(vii) By cases on the form of ρ. If $\rho \equiv \mathtt{pro}\, t.\langle\!\langle R \rangle\!\rangle \lhd \langle M \rangle$, we have the thesis by point (i) and Lemma 6.3.(i) (Well-formed types). If $\rho \equiv t \lhd \langle M \rangle$, we reason by cases on the form of ξ: if $\xi \equiv \mathtt{pro}\, t.\langle\!\langle S \rangle\!\rangle \lhd \langle N \rangle$, then we have the thesis by point (iii) and the validity of thesis for pro-types; if $\xi \equiv t' \lhd \langle N \rangle$, then we have the thesis by point (ii).

(viii) By cases on the form of ρ, using either point (i) or point (iii).

(ix) By cases on the form of ρ, using either point (i) or point (ii) and Lemma 6.3.(ii) (Well-formed types). □

Lemma 6.6 *(Match Weakening)*

(i) *If* $\Gamma, t <\!\!\# \, \rho, \Gamma' \vdash A$, $\Gamma \vdash \xi <\!\!\# \, \rho$ *and* ξ *is a* pro*-type, then* $\Gamma, t <\!\!\# \, \xi, \Gamma' \vdash A$.

(ii) *If* $\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma \rangle\!\rangle \lhd \langle M \rangle : T$, *then* $\Gamma, t <\!\!\# \, \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma \rangle\!\rangle \lhd \langle M \rangle \vdash \sigma : T$.

(i) By structural induction on the derivation Δ of $\Gamma, t <\!\!\# \, \rho, \Gamma' \vdash A$. The only case where the inductive hypothesis cannot be applied is when Γ' is empty and the last rule in Δ is a rule increasing the length of the context (context rule). In this latter case $\Gamma, t <\!\!\# \, \rho \vdash ok$ has been derived via the rule $(Cont{-}t)$. We have therefore $t \notin dom(\Gamma)$ and, by the second hypothesis and Lemma 6.4, $\Gamma \vdash \xi : T$, from which we derive the thesis using the $(Cont{-}t)$ rule.

For all the other cases but one the application of the inductive hypothesis and the derivation of the thesis is immediate, since the last rule in Δ does not use the hypothesis $t <\!\!\# \, \rho$ in the context. The only rule that can use this hypothesis is $(Match{-}Var)$. If the last rule in Δ is $(Match{-}Var)$, then $\Gamma, t <\!\!\# \, \rho, \Gamma' \vdash t \lhd \langle M \rangle <\!\!\# \, \rho'$ derives from the premise $\Gamma, t <\!\!\# \, \rho, \Gamma' \vdash \rho \lhd \langle M \rangle <\!\!\# \, \rho'$. By inductive hypothesis, we have $\Gamma, t <\!\!\# \, \xi, \Gamma' \vdash \rho \lhd \langle M \rangle <\!\!\# \, \rho'$. Moreover, from $\Gamma \vdash \xi <\!\!\# \, \rho$ and the Weakening Lemma 6.2, we derive $\Gamma, t <\!\!\# \, \xi, \Gamma' \vdash \xi <\!\!\# \, \rho$, from which, by Lemma 6.5(vii), $\Gamma, t <\!\!\# \, \xi, \Gamma' \vdash \xi \lhd \langle M \rangle <\!\!\# \, \rho \lhd \langle M \rangle$. Finally, by transitivity of matching we deduce $\Gamma, t <\!\!\# \, \xi, \Gamma' \vdash \xi \lhd \langle M \rangle <\!\!\# \, \rho'$, and by an application of the $(Match{-}Var)$ rule we obtain the thesis.

(ii) First observe that there exists $S \subseteq R$ such that $\Gamma, t <\!\!\# \, \mathtt{pro}\, t.\langle\!\langle S \rangle\!\rangle \vdash \sigma : T$. In fact, by Lemma 6.3.(i) (Well-formed types), we have $\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R, m : \sigma \rangle\!\rangle : T$, that can only be derived by an application of the $(Type{-}Pro)$ rule; therefore we have either our goal or $\Gamma, t <\!\!\# \, \mathtt{pro}\, t.\langle\!\langle R', \mathtt{m} : \sigma \rangle\!\rangle \vdash \sigma' : T$, for a suitable R' such that $R \equiv R', m' : \sigma'$. By Lemma 6.1.(iii) (Sub-derivation), it follows that $\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R', m : \sigma \rangle\!\rangle : T$, and, by repeating these last two arguments, we conclude the existence of S. Now, from $\Gamma, t <\!\!\# \, \mathtt{pro}\, t.\langle\!\langle S \rangle\!\rangle \vdash \sigma : T$, by using Lemma 6.1.(iii) (Sub-derivation), the $(Match{-}Pro)$ rule and point (i), we have the thesis. □

Proposition 6.7 *(Substitution)*

(i) *If* $\Gamma, x : \tau, \Gamma' \vdash A$ *and* $\Gamma \vdash e : \tau$, *then* $\Gamma, \Gamma' \vdash A[e/x]$.

(ii) *If* $\Gamma, t <\!\!\# \rho, \Gamma', \Gamma'' \vdash A$ *and* $\Gamma, t <\!\!\# \rho, \Gamma' \vdash \xi <\!\!\# \rho$, *then* $\Gamma, t <\!\!\# \rho, \Gamma', \Gamma''[\xi/t] \vdash A[\xi/t]$.

(iii) *If* $\Gamma, t <\!\!\# \rho, \Gamma' \vdash A$ *and* $\Gamma \vdash \xi <\!\!\# \rho$, *then* $\Gamma, \Gamma'[\xi/t] \vdash A[\rho/t]$.

(i) By induction on the derivation Δ of $\Gamma, x : \tau, \Gamma' \vdash A$. The cases where the inductive hypothesis cannot be immediately applied are the ones where the last rule in Δ is a context rule and Γ' is empty. In this case $\Gamma, x : \tau \vdash ok$ is derived from $\Gamma \vdash \tau : T$, from which, by Lemma 6.1.(i) (Sub-derivation), we have the thesis.

The only not trivial case is the one where the last rule in Δ is (Var) and the variable x coincides with the one dealt with by the rule. In this case the conclusion $\Gamma, x : \tau, \Gamma' \vdash x : \tau$ derives from the premise $\Gamma, x : \tau, \Gamma' \vdash ok$, and so, by induction, $\Gamma, \Gamma' \vdash ok$. By the second hypothesis $\Gamma \vdash e : \tau$ and Lemma 6.2 (Weakening), we deduce $\Gamma, \Gamma' \vdash e : \tau$, which coincides with our thesis. All the remaining cases can be easily proved by applying the inductive hypothesis.

(ii) By induction on the derivation Δ of $\Gamma, t <\!\!\# \rho, \Gamma', \Gamma'' \vdash A$. As for the previous point, the only cases where the inductive hypothesis cannot be applied are the ones where the last rule in Δ is a context rule (Γ', Γ'' are empty), but these cases are trivial, since the hypothesis coincides with the thesis.

About the remaining cases, the only not trivial one is when the last rule in Δ is $(Match{-}Var)$ (the only rule that can use the judgment $t <\!\!\# \rho$, contained in the context) and the type variable t coincides with the one dealt with by the rule. In this case the conclusion $\Gamma, t <\!\!\# \rho, \Gamma', \Gamma'' \vdash t \triangleleft \langle M \rangle <\!\!\# \rho'$ derives from the premise $\Gamma, t <\!\!\# \rho, \Gamma', \Gamma'' \vdash \rho \triangleleft \langle M \rangle <\!\!\# \rho'$; then, by inductive hypothesis, $\Gamma, t <\!\!\# \rho, \Gamma', \Gamma''[\xi/t] \vdash (\rho \triangleleft \langle M \rangle <\!\!\# \rho')[\xi/t])$. By the side condition on $(Cont - t)$, t cannot be free in ρ, and, by Lemma 6.5 (i), t cannot be free in ρ' either, hence the above judgment can be written as $\Gamma, t <\!\!\# \rho, \Gamma', \Gamma''[\xi/t] \vdash \rho \triangleleft \langle M \rangle <\!\!\# \rho'$. On the other hand, from the hypothesis $\Gamma, t <\!\!\# \rho, \Gamma' \vdash \xi <\!\!\# \rho$, we can derive $\Gamma, t <\!\!\# \rho, \Gamma', \Gamma''[\xi/t] \vdash \xi \triangleleft \langle M \rangle <\!\!\# \rho \triangleleft \langle M \rangle$ by Lemma 6.2.(ii) (Weakening) and Lemma 6.5.(vii), and from the transitivity of matching (Lemma 6.5.(v)) we have $\Gamma, t <\!\!\# \rho, \Gamma'\Gamma''[\xi/t] \vdash \xi \triangleleft \langle M \rangle <\!\!\# \rho'$, which is our thesis.

(iii) By the previous point we can derive $\Gamma, t <\!\!\# \rho, \Gamma'[\xi/t] \vdash A[\rho/t]$. Now, via an immediate induction one can prove that if $\Gamma, t <\!\!\# \rho, \Gamma' \vdash A$ and t does not appear free in Γ' and A, then $\Gamma, \Gamma' \vdash A$. From this we have immediately the thesis. □

Proposition 6.8 *(Types of expressions are well-formed)*
If $\Gamma \vdash e : \tau$, *then* $\Gamma \vdash \tau : T$.

By structural induction on the derivation Δ of $\Gamma \vdash e : \tau$. In this proof we need to consider explicitly all the possible cases for the last rule in Δ; each case is quite simple but it needs specific arguments.

If the last rule in Δ is $(Const)$, we derive the thesis via the $(Type{-}Const)$ rule. If the last rule is (Var), we use Lemma 6.1.(ii) (Sub-derivation) and Lemma 6.2.(i) (Weakening). For the rule (Abs) one applies the inductive hypothesis, Lemma 6.1.(ii) (Sub-derivation), Lemma 6.7.(i) (Substitution), and the $(Type{-}Arrow)$ rule. For the rule $(Appl)$ one applies the inductive hypothesis, obtaining $\Gamma \vdash \sigma \to \tau : T$; this judgment can only be derived through the $(Type{-}Arrow)$ rule, whose second premise is just the thesis.

Now we consider the rules for object terms. The thesis is trivial for the rules (*Empty*), (*Pre−Extend*) and (*Override*). In the case of (*Extend*), $\Gamma \vdash \langle e_1 \prec\!\!\!- \mathtt{m} = e_2\rangle : \rho \triangleleft \mathtt{m}$ is derived from $\Gamma \vdash \rho <\!\!\# \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma\rangle\!\rangle \triangleleft\langle M\rangle$; by Proposition 6.4 and Lemma 6.3.(i), we have $\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma\rangle\!\rangle \triangleleft\langle M, \mathtt{m}\rangle : T$; by Lemma 6.5.(vii), we have $\Gamma \vdash \rho \triangleleft \mathtt{m} <\!\!\# \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma\rangle\!\rangle \triangleleft\langle M, \mathtt{m}\rangle$, and so we conclude by Proposition 6.4.

The two remaining cases are more complex.

(*Send*) In this case we assume that $\Gamma \vdash e \Leftarrow \mathtt{m} : \sigma[\rho/t]$ is derived from the $\Gamma \vdash \rho <\!\!\# \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma\rangle\!\rangle \triangleleft\langle M, \mathtt{m}\rangle$ premise, from which, by Proposition 6.4, we have $\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma\rangle\!\rangle \triangleleft\langle M, \mathtt{m}\rangle : T$ and, in turn, $\Gamma, t <\!\!\# \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma\rangle\!\rangle \triangleleft\langle M, \mathtt{m}\rangle \vdash \sigma : T$ by Lemma 6.6.(ii); finally, by Proposition 6.7.(iii) (Substitution), we can conclude $\Gamma \vdash \sigma[\rho/t] : T$.

(*Select*) In this case we assume that $\Gamma \vdash Sel(e_1, \mathtt{m}, e_2) : \sigma[(\rho \triangleleft\langle M\rangle)/t]$ is derived from the premises $\Gamma, s <\!\!\# \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma\rangle\!\rangle \vdash e_2 : s \to (s \triangleleft\langle N\rangle)$ and $\Gamma \vdash \rho <\!\!\# \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma\rangle\!\rangle \triangleleft\langle M, \mathtt{m}\rangle$. By inductive hypothesis, we have $\Gamma, s <\!\!\# \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma\rangle\!\rangle \vdash s \to (s \triangleleft\langle N\rangle) : T$; by Proposition 6.7.(iii) (Substitution), we derive $\Gamma \vdash \rho \to (\rho \triangleleft\langle N\rangle) : T$ and, since this latter judgment can be only obtained via the (Type-Arrow) rule, we deduce $\Gamma \vdash \rho \triangleleft\langle N\rangle : T$. Moreover, we have $\Gamma \vdash \rho \triangleleft\langle N\rangle <\!\!\# \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma\rangle\!\rangle \triangleleft\langle M, \mathtt{m}\rangle$ by case analysis and Lemma 6.5.(i)-(iii), from which the thesis follows by Lemma 6.6.(ii) and Proposition 6.7.(iii) (Substitution), i.e. $\Gamma \vdash \sigma[(\rho \triangleleft\langle M\rangle)/t] : T$. □

Finally, we can state the key Subject Reduction property for our type system.

Theorem 6.9 *(Subject Reduction) If* $\Gamma \vdash e : \tau$ *and* $e \xrightarrow{ev} e'$*, then* $\Gamma \vdash e' : \tau$.

We prove that the type is preserved by each of the four reduction rules (*Beta*), (*Select*), (*Success*) and (*Next*).

(*Beta*) The derivation Δ of $\Gamma \vdash (\lambda x.e_1)e_2 : \tau$ needs to terminate with a rule (*Appl*), deriving $\Gamma \vdash (\lambda x.e_1)e_2 : \tau'$, potentially followed by some applications of (*Pre−Extend*). Let the premises of (*Abs*) be $\Gamma \vdash (\lambda x.e_1) : \upsilon \to \tau'$ and $\Gamma \vdash e_2 : \upsilon$ for a suitable υ; in turn, the first judgment has to be deduced from $\Gamma, x : \upsilon \vdash e_1 : \tau'$ via the (*Abs*) rule. By Proposition 6.7.(ii) (Substitution), we conclude $\Gamma \vdash (e_1 : \tau)[e_2/x] \equiv e_1[e_2/x] : \tau$, and, by some potential applications of (*Pre−Extend*), we have the thesis.

(*Select*) The derivation Δ of $\Gamma \vdash e \Leftarrow \mathtt{m} : \tau$ needs to terminate with a rule (*Send*), deriving $\Gamma \vdash e \Leftarrow \mathtt{m} : \sigma[\rho/t]$, potentially followed by some applications of (*Pre−Extend*). The premises of the (*Send*) rule are $\Gamma \vdash e : \rho$ and $\Gamma \vdash \rho <\!\!\# \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma\rangle\!\rangle \triangleleft\langle M, \mathtt{m}\rangle$. From this latter judgment, by Lemma 6.4 (Matching has well-formed types) and the rules (*Cont−t*), (*Match−Pro*), (*Match−Var*), (*Type−Extend*), (*Cont−x*), (*Var*), and (*Abs*), one can derive $\Gamma, s <\!\!\# \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma\rangle\!\rangle \triangleleft\langle M, \mathtt{m}\rangle \vdash \lambda x.\, x : s \to s$. From the above premises, by applying the (*Select*) rule, we derive $\Gamma \vdash Sel(e, \mathtt{m}, \lambda x.\, x) : \sigma[\rho/t]$ and, by some potential applications of (*Pre−Extend*), we have the thesis.

(*Success*) The derivation Δ of $\Gamma \vdash Sel(\langle e_1 \prec\!\!\!- \mathtt{m} = e_2\rangle, \mathtt{m}, e) : \tau$ needs to terminate with a rule (*Select*), deriving $\Gamma \vdash Sel(\langle e_1 \prec\!\!\!- \mathtt{m} = e_2\rangle, \mathtt{m}, e) : \sigma[(\rho \triangleleft\langle N\rangle)/t]$, potentially followed by some applications of (*Pre−Extend*). The premises of the (*Select*) rule are:

$$\Gamma \vdash \langle e_1 \prec\!\!\!- \mathtt{m} = e_2\rangle : \rho \tag{2}$$

$$\Gamma \vdash \rho <\!\!\# \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma\rangle\!\rangle \triangleleft\langle M, \mathtt{m}\rangle \tag{3}$$

$$\Gamma, s <\!\!\# \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma\rangle\!\rangle \triangleleft\langle M, \mathtt{m}\rangle \vdash e : s \to (s \triangleleft\langle N\rangle) \tag{4}$$

From the judgments (3) and (4), by Substitution, we have $\Gamma \vdash e : \rho \to (\rho \lhd \langle N \rangle)$; from this judgment and (2), by the rule (*Appl*), we derive:

$$\Gamma \vdash e\langle e_1 \prec\!\!\!\leftarrow \mathtt{m} = e_2\rangle : \rho \lhd \langle N \rangle \tag{5}$$

The judgment (2) can only be derived using either the (*Extend*) rule or the (*Override*) one, potentially followed by some applications of (*Pre−Extend*). Here we will consider only the case where the (*Extend*) rule is applied, since the case for (*Override*) can be proved using the same arguments, with the only difference that in some points the proof is simpler. So, assume that the (*Extend*) rule derives $\Gamma \vdash \langle e_1 \prec\!\!\!\leftarrow \mathtt{m} = e_2\rangle : \rho' \lhd \mathtt{m}$ by using as premises:

$$\Gamma \vdash e_1 : \rho'$$

$$\Gamma \vdash \rho' <\!\# \, \mathtt{pro}\, t.\langle\!\langle R', \mathtt{m} : \sigma'\rangle\!\rangle \lhd \langle M'\rangle \tag{6}$$

$$\Gamma, t <\!\# \, \mathtt{pro}\, t.\langle\!\langle R', \mathtt{m} : \sigma'\rangle\!\rangle \lhd \langle M', \mathtt{m}\rangle \vdash e_2 : t \to \sigma' \tag{7}$$

By inspection on the (*Pre−Extend*) rule, we can readily derive $\Gamma \vdash \rho <\!\# \, \rho' \lhd \mathtt{m}$. From judgment (6), by Lemma 6.5.(vii), we have $\Gamma \vdash \rho' \lhd \mathtt{m} <\!\# \, \mathtt{pro}\, t.\langle\!\langle R', \mathtt{m} : \sigma'\rangle\!\rangle \lhd \langle M', \mathtt{m}\rangle$, and, by transitivity of matching, we derive $\Gamma \vdash \rho <\!\# \, \mathtt{pro}\, t.\langle\!\langle R', \mathtt{m} : \sigma'\rangle\!\rangle \lhd \langle M', \mathtt{m}\rangle$. From this judgment and (3), by Lemma 6.5.(vi)(Matching uniqueness), it follows that $\sigma \equiv \sigma'$. Moreover, by Lemma 6.5.(ix), we have $\Gamma \vdash \rho \lhd \langle N\rangle <\!\# \, \rho$ and, by transitivity of matching, $\Gamma \vdash \rho \lhd \langle N\rangle <\!\# \, \mathtt{pro}\, t.\langle\!\langle R', \mathtt{m} : \sigma\rangle\!\rangle \lhd \langle M', \mathtt{m}\rangle$. From this judgment and (7), by Substitution, we have, $\Gamma \vdash e_2 : \rho \lhd \langle N\rangle \to \sigma[(\rho \lhd \langle N\rangle)/t]$, and from this and (5), by the rule (*Appl*), we have $\Gamma \vdash e_2\ (e\langle e_1 \prec\!\!\!\leftarrow \mathtt{m} = e_2\rangle) : \sigma[(\rho \lhd \langle N\rangle)/t]$; then, by some potential applications of the (*Pre−Extend*) rule, we have the thesis.

(*Next*) As argued for the rule (*Success*), the derivation Δ of $\Gamma \vdash \mathit{Sel}(\langle e_1 \prec\!\!\!\leftarrow \mathtt{n} = e_2\rangle, \mathtt{m}, e) : \tau$ needs to terminate with a rule (*Select*), deriving $\Gamma \vdash \mathit{Sel}(\langle e_1 \prec\!\!\!\leftarrow \mathtt{n} = e_2\rangle, \mathtt{m}, e) : \sigma[(\rho \lhd \langle N\rangle)/t]$, potentially followed by some applications of (*Pre−Extend*). Let the premises of the (*Select*) rule be:

$$\Gamma \vdash \langle e_1 \prec\!\!\!\leftarrow \mathtt{n} = e_2\rangle : \rho \tag{8}$$

$$\Gamma \vdash \rho <\!\# \, \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma\rangle\!\rangle \lhd \langle M, \mathtt{m}\rangle \tag{9}$$

$$\Gamma, s <\!\# \, \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma\rangle\!\rangle \lhd \langle M, \mathtt{m}\rangle \vdash e : s \to (s \lhd \langle N\rangle) \tag{10}$$

The judgment (8) can only be derived using either the (*Extend*) rule or the (*Override*) one, potentially followed by some applications of (*Pre−Extend*). Here we only consider the case where both (*Extend*) and (*Pre−Extend*) are applied; the other case can be derived with a similar pattern.

Since (*Pre−Extend*) has been applied and (8) holds, ρ must be in the form $\mathtt{pro}\, t.\langle\!\langle R', \mathtt{m} : \sigma, \mathtt{n} : \sigma'\rangle\!\rangle \lhd \langle M', \mathtt{m}, \mathtt{n}\rangle$. So, let (8) be derived by (*Pre − Extend*) from:

$$\Gamma \vdash \langle e_1 \prec\!\!\!\leftarrow \mathtt{n} = e_2\rangle : \mathtt{pro}\, t.\langle\!\langle R'', \mathtt{m} : \sigma, \mathtt{n} : \sigma'\rangle\!\rangle \lhd \langle M', \mathtt{m}, \mathtt{n}\rangle$$

(where $R'' \subseteq R'$), which, in turn, is derived by the (*Extend*) rule from the premises:

$$\Gamma \vdash e_1 : \mathtt{pro}\, t.\langle\!\langle R'', \mathtt{m} : \sigma, \mathtt{n} : \sigma'\rangle\!\rangle \lhd \langle M', \mathtt{m}\rangle \tag{11}$$

$$\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R'', \mathtt{m} : \sigma, \mathtt{n} : \sigma'\rangle\!\rangle \lhd \langle M', \mathtt{m}\rangle <\!\# \, \mathtt{pro}\, t.\langle\!\langle R''', \mathtt{n} : \sigma'\rangle\!\rangle \lhd \langle M'''\rangle \tag{12}$$

$$\Gamma \vdash t <\!\# \, \mathtt{pro}\, t.\langle\!\langle R''', \mathtt{n} : \sigma'\rangle\!\rangle \lhd \langle M''', \mathtt{n}\rangle \vdash e_2 : t \to \sigma' \tag{13}$$

Moreover, let ρ' indicate the type $\mathtt{pro}\, t.\langle\!\langle R', \mathtt{m} : \sigma, \mathtt{n} : \sigma' \rangle\!\rangle \lhd \langle M', \mathtt{m} \rangle$, i.e. $\rho \equiv \rho' \lhd n$. From the judgment (11), by the $(Pre{-}Extend)$ rule, we can derive:

$$\Gamma \vdash e_1 : \rho' \tag{14}$$

By the $(Match{-}Pro)$ rule, we have $\Gamma \vdash \rho' \lhd n <\!\!\# \mathtt{pro}\, t.\langle\!\langle R'', \mathtt{m} : \sigma, \mathtt{n} : \sigma' \rangle\!\rangle \lhd \langle M', \mathtt{m} \rangle$; from this judgment, (12) and (13), by transitivity of matching and the Weakening Lemma, we have $\Gamma, t <\!\!\# \rho' \lhd \mathtt{n} \vdash e_2 : t \to \sigma'$. From this judgment, using the $(Extend)$ rule, one can easily derive $\Gamma, s <\!\!\# \rho', x : s \vdash \langle x \prec\!\!\!\leftarrow \mathtt{n} = e_2 \rangle : s \lhd \mathtt{n}$. Using (9), $(Match - Var)$, and transitivity one can derive $\Gamma, s <\!\!\# \rho' \vdash s \lhd \mathtt{n} <\!\!\# \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma \rangle\!\rangle \lhd \langle M, \mathtt{m} \rangle$. From this latter judgment and (10), by Substitution, we obtain $\Gamma, s <\!\!\# \rho' \vdash e : (s \lhd \mathtt{n}) \to s \lhd \mathtt{n} \lhd \langle N \rangle$. From the last and the last but two judgments, by $(Appl)$ and (Abs) rules, we have:

$$\Gamma, s <\!\!\# \rho' \vdash \lambda x.e\langle x \prec\!\!\!\leftarrow \mathtt{n} = e_2 \rangle : s \to s \lhd \mathtt{n} \lhd \langle N \rangle$$

This judgment, together with (14), allows to apply the $(Select)$ rule, thus deriving:

$$\Gamma \vdash Sel(e_1, \mathtt{m}, \lambda x.e\langle x \prec\!\!\!\leftarrow \mathtt{n} = e_2 \rangle) : \sigma[(\rho' \lhd \mathtt{n} \lhd \langle N \rangle)/t]$$

Therefore we can readily obtain the thesis. □

We conclude by deducing the Type Soundness theorem: it will guarantee, among other, that every closed and well typed expression will not produce the `message-not-found` runtime error. This error arises whenever we search for a message `m` into an expression that does not reduce to an object which has the method `m` in its interface.

Definition 6.10 *Define the set of* `wrong` *terms as follows:*

$$\mathtt{wrong} \quad ::= \quad Sel(\langle\,\rangle, \mathtt{m}, e) \mid Sel((\lambda x.e), \mathtt{m}, e') \mid Sel(c, \mathtt{m}, e)$$

By a direct inspection of the typing rules for terms, one can immediately see that `wrong` cannot be typed. Hence, the Type Soundness follows as a corollary of the Subject Reduction theorem.

Corollary 6.11 *(Type Soundness) If* $\varepsilon \vdash e : \tau$*, then* $e \overset{ev}{\not\twoheadrightarrow} C[\mathtt{wrong}]$*, where* $C[\,]$ *is a generic context in* $\lambda\mathcal{O}bj+$*, i.e. a term with an "hole" inside it.*

6.1 Soundness of the Type System with Subsumption

The proof of the Subject Reduction Theorem concerning the type assignment system with subsumption is quite similar to the corresponding proof for the plain type system. In particular, all the preliminary lemmas and their corresponding proofs remain almost the same; only the final proof of Theorem 6.9 needs to be modified significantly. Since that, we do not present here the whole proofs of the preliminary lemmas, but we just remark the points where new arguments are needed.

In fact, Lemmas 6.1 (Sub-derivation), 6.2 (Weakening), 6.4 (Matching has well-formed types), 6.7 (Substitution), 6.8 (Types of expressions are well-formed) are valid also for the type assignment with subsumption. Conversely, we need to rephrase Lemmas 6.3 (Well-formed types), 6.5 (Matching), 6.6 (Match Weakening) as follows.

In Lemma (Well-formed types) 6.3, the point (ii) needs to be rewritten as:

(ii) $\Gamma \vdash t \lhd \langle M \rangle : T$ if and only if Γ contains $t <\!\!\# \mathtt{pro}\, t.\langle\!\langle R \rangle\!\rangle \lhd \langle N \rangle$ or $t <\!\!\# \mathtt{obj}\, t.\langle\!\langle R \rangle\!\rangle \lhd \langle N \rangle$, with $\{M\} \subseteq \mathcal{M}(R)$.

In Lemma (Matching) 6.5, the point (vi) needs to be rewritten as:

(vi) (Uniqueness) if $\Gamma \vdash \rho <\!\!\# \, \mathtt{obj}\, t.\langle\!\langle R, m : \sigma \rangle\!\rangle$ and $\Gamma \vdash \rho <\!\!\# \, \mathtt{obj}\, t.\langle\!\langle R', m : \sigma' \rangle\!\rangle$, then $\sigma \equiv \sigma'$.

Moreover, the following points need to be added:

(i') $\Gamma \vdash \mathtt{obj}\, t.\langle\!\langle R \rangle\!\rangle \lhd \langle M \rangle <\!\!\# \, \rho$ if and only if $\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R \rangle\!\rangle \lhd \langle M \rangle : T$ and $\Gamma \vdash \rho : T$ and $\rho \equiv \mathtt{obj}\, t.\langle\!\langle R' \rangle\!\rangle \lhd \langle M' \rangle$, with $R' \subseteq R$ and $\{M'\} \subseteq \{M\}$.

(iii') $\Gamma \vdash t \lhd \langle M \rangle <\!\!\# \, \mathtt{obj}\, t.\langle\!\langle R \rangle\!\rangle \lhd \langle N \rangle$ if and only if Γ contains $t <\!\!\# \, \mathtt{obj}\, t.\langle\!\langle R' \rangle\!\rangle \lhd \langle N' \rangle$ or $t <\!\!\# \, \mathtt{pro}\, t.\langle\!\langle R' \rangle\!\rangle \lhd \langle N' \rangle$, with $R \subset R'$ and $\{N\} \subset \{M \cup N'\}$.

(viii') If $\Gamma \vdash \rho \lhd \mathtt{m} <\!\!\# \, \mathtt{obj}\, t.\langle\!\langle R \rangle\!\rangle \lhd \langle M \rangle$, then $\Gamma \vdash \rho <\!\!\# \, \mathtt{obj}\, t.\langle\!\langle R \rangle\!\rangle \lhd \langle M - \mathtt{m} \rangle$.

In Lemma 6.6 (Match Weakening), the point (ii) needs to be rewritten as:

(ii) If $\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma \rangle\!\rangle \lhd \langle M \rangle : T$ or $\Gamma \vdash \mathtt{obj}\, t.\langle\!\langle R, \mathtt{m} : \sigma \rangle\!\rangle \lhd \langle M \rangle : T$, then $\Gamma, t <\!\!\# \, \mathtt{obj}\, t.\langle\!\langle R, \mathtt{m} : \sigma \rangle\!\rangle \lhd \langle M \rangle \vdash \sigma : T$.

A new lemma, stating some elementary properties of types with covariant variables and rigid types is necessary.

Lemma 6.12 *(Covariant variables and Rigid types)*

(i) If t is covariant in σ and $\Gamma \vdash \sigma : Rgd$ and $\Gamma \vdash \xi <\!\!\# \, \rho$, then $\Gamma \vdash \sigma[\xi/t] <\!\!\# \, \sigma[\rho/t]$.

(ii) If $\Gamma \vdash \sigma : Rgd$ and $\Gamma \vdash \rho : Rgd$, then $\Gamma \vdash \sigma[\rho/t] : Rgd$.

As already remarked, the Subject Reduction for the type assignment system with subsumption has the usual formulation, but it needs a more complex proof.

Theorem 6.13 *(Subject Reduction, full $\lambda\mathcal{O}bj+$) If $\Gamma \vdash e : \tau$ and $e \stackrel{ev}{\to} e'$, then $\Gamma \vdash e' : \tau$.*

Similarly to Theorem 6.9, we prove that the type is preserved by each of the four reduction rules (*Beta*), (*Select*), (*Success*) and (*Next*). In the present case we have the extra difficulty of considering the possible application of the (*Subsume*) rule.

(*Beta*) The derivation of $\Gamma \vdash (\lambda x.e_1)e_2 : \tau$ needs to terminate with a rule (*Appl*), deriving $\Gamma \vdash (\lambda x.e_1)e_2 : \xi$, potentially followed by some applications of the ($Pre{-}Extend$) and (*Subsume*) rules. The premises of (*Appl*) must be $\Gamma \vdash (\lambda x.e_1) : \upsilon \to \xi$ and $\Gamma \vdash e_2 : \upsilon$ for a suitable υ; in turn, the first judgment has to be deduced via the (*Abs*) rule, potentially followed by (*Subsume*). Let $\Gamma \vdash (\lambda x.e_1) : \upsilon' \to \xi'$ be the conclusion of the (*Abs*) rule, and:

$$\Gamma, x : \upsilon' \vdash e_1 : \xi' \tag{15}$$

its premise. Since the (*Subsume*) rule has been applied, we have $\Gamma \vdash \upsilon' \to \xi' <\!\!\# \, \upsilon \to \xi$ and $\Gamma \vdash \upsilon \to \xi : Rgd$, therefore $\Gamma \vdash \upsilon <\!\!\# \, \upsilon'$ and $\Gamma \vdash \upsilon' : Rgd$ and $\Gamma \vdash \xi' <\!\!\# \, \xi$ and $\Gamma \vdash \xi : Rdg$. Using these judgments and (15), it is not difficult to prove, by structural induction, that $\Gamma, x : \upsilon \vdash e_1 : \xi'$. By substitution, we have then $\Gamma \vdash e_1[e_2/x] : \xi'$, and, by the (*Subsume*) rule, $\Gamma \vdash e_1[e_2/x] : \xi$, from which the thesis.

(*Select*) This case works as for the system without subsumption.

(*Success*) Repeating the arguments used for the Subject Reduction 6.9, we can say that the derivation Δ of $\Gamma \vdash Sel(\langle e_1 \prec\!\!\!\leftarrow \mathtt{m} = e_2 \rangle, \mathtt{m}, e) : \tau$ needs to terminate with a rule

$(Select)$, deriving $\Gamma \vdash Sel(\langle e_1 \prec\!\!\!\leftarrow \mathtt{m} = e_2\rangle, \mathtt{m}, e) : \sigma[(\rho \triangleleft \langle N\rangle)/t]$, potentially followed by some applications of the $(Pre{-}Extend)$ and $(Subsume)$ rules. The premises of the $(Select)$ rule are:

$$\Gamma \vdash \langle e_1 \prec\!\!\!\leftarrow \mathtt{m} = e_2\rangle : \rho \tag{16}$$

$$\Gamma \vdash \rho <\!\!\# \,\mathtt{obj}\, t.\langle\!\langle R, \mathtt{m} : \sigma\rangle\!\rangle \triangleleft \langle M, \mathtt{m}\rangle \tag{17}$$

$$\Gamma, s <\!\!\# \,\mathtt{obj}\, t.\langle\!\langle R, \mathtt{m} : \sigma\rangle\!\rangle \triangleleft \langle M, \mathtt{m}\rangle \vdash e : s \to (s \triangleleft \langle N\rangle) \tag{18}$$

If the judgment (16) is not derived by an application of the $(Subsume)$ rule, it is possible to repeat the arguments used for the type system without subsumption. Hence, we consider only the case where (16) is derived by a single application of the $(Subsume)$ rule. Notice that it is sufficient to consider a single application, since consecutive (multiple) applications of $(Subsume)$ can always be compacted in a single one. So, let the premises of $(Subsume)$ be:

$$\Gamma \vdash \langle e_1 \prec\!\!\!\leftarrow \mathtt{m} = e_2\rangle : \xi \tag{19}$$

$$\Gamma \vdash \xi <\!\!\# \rho \tag{20}$$

$$\Gamma \vdash \rho : Rgd \tag{21}$$

From the judgments (17), (20) and (18), by transitivity and Substitution, we have $\Gamma \vdash e : \xi \to (\xi \triangleleft \langle N\rangle)$. From this and (19), by the $(Appl)$ rule, we have:

$$\Gamma \vdash e\langle e_1 \prec\!\!\!\leftarrow \mathtt{m} = e_2\rangle : \xi \triangleleft \langle N\rangle \tag{22}$$

By repeating the same arguments used for the previous version of the Theorem 6.9 (case analysis on the derivation of (19)), we can prove that:

$$\Gamma \vdash e_2\ (e\langle e_1 \prec\!\!\!\leftarrow \mathtt{m} = e_2\rangle) : \sigma[(\xi \triangleleft \langle N\rangle)/t]$$

Moreover, from (17) and (21) we can derive that t is covariant in σ and $\Gamma \vdash \sigma : Rdg$. By Lemma 6.12, it follows that $\Gamma \vdash \sigma[(\xi \triangleleft \langle N\rangle)/t] <\!\!\# \sigma[(\rho \triangleleft \langle N\rangle)/t]$ and $\Gamma \vdash \sigma[(\rho \triangleleft \langle N\rangle)/t] : Rgd$ and so, by an application of the $(Subsume)$ rule, we have $\Gamma \vdash e_2\ (e\langle e_1 \prec\!\!\!\leftarrow \mathtt{m} = e_2\rangle) : \sigma[(\rho \triangleleft \langle N\rangle)/t]$, and from this the thesis.

$(Next)$ Also in this case we can just repeat the arguments used for the former version of the theorem, thus taking care of the possible applications of the $(Subsume)$ rule. So, we argue that the derivation Δ of $\Gamma \vdash Sel(\langle e_1 \prec\!\!\!\leftarrow \mathtt{n} = e_2\rangle, \mathtt{m}, e) : \tau$ needs to terminate with a rule $(Select)$, deriving $\Gamma \vdash Sel(\langle e_1 \prec\!\!\!\leftarrow \mathtt{n} = e_2\rangle, \mathtt{m}, e) : \sigma[(\rho \triangleleft \langle N\rangle)/t]$, potentially followed by some applications of the $(Pre{-}Extend)$ and $(Subsume)$ rules. Let the premises of the $(Select)$ rule be:

$$\Gamma \vdash \langle e_1 \prec\!\!\!\leftarrow \mathtt{n} = e_2\rangle : \rho \tag{23}$$

$$\Gamma \vdash \rho <\!\!\# \,\mathtt{obj}\, t.\langle\!\langle R, \mathtt{m} : \sigma\rangle\!\rangle \triangleleft \langle M, \mathtt{m}\rangle \tag{24}$$

$$\Gamma, s <\!\!\# \,\mathtt{obj}\, t.\langle\!\langle R, \mathtt{m} : \sigma\rangle\!\rangle \triangleleft \langle M, \mathtt{m}\rangle \vdash e : s \to (s \triangleleft \langle N\rangle) \tag{25}$$

If the judgment (23) is not derived by an application of the $(Subsume)$ rule, it is possible to repeat the arguments used in the proof of the Theorem 6.9. Then, we address only the case where (16) is derived by a single application of the $(Subsume)$ rule, having as premises:

$$\Gamma \vdash \langle e_1 \prec\!\!\!\leftarrow \mathtt{m} = e_2\rangle : \xi \tag{26}$$

$$\Gamma \vdash \xi \lessdot\# \rho \tag{27}$$

$$\Gamma \vdash \rho : Rgd \tag{28}$$

From these judgments, by repeating the same steps argued for the proof without subsumption (case analysis on the derivation of the judgment (26)), we deduce:

$$\Gamma \vdash Sel(e_1, \mathtt{m}, \lambda x.e\langle x \prec\!\!\!\!\!\!- \mathtt{n} = e_2\rangle) : \sigma[(\xi \lhd \mathtt{n} \lhd \langle N \rangle)/t]$$

Therefore, by repeating the final arguments used for the (*Success*) case, i.e. by applying the Lemma 6.12 and the (*Subsume*) rule, we can derive the thesis. □

7 Object reclassification

In class-based, object-oriented programming, the possibility of changing at runtime the class membership of an object while retaining its identity is known as *(dynamic) object reclassification*. One major contribution to the development of reclassification features has produced the Java-like `Fickle` language, in its incremental versions [DDDG01, DDDG02, DDG03].

In this section, we show how the self-inflicted extension primitive provided by our calculus may be used to mimic the mechanisms implemented in `Fickle`.

We proceed, suggestively, by working out a case study: first we write an example in `Fickle` which illustrates the essential ingredients of the reclassification, then we devise and discuss the possibilities of its encoding in $\lambda\mathcal{O}bj+$.

7.1 Reclassification in `Fickle`

`Fickle` is an imperative, class-based, strongly-typed language, where classes are types and subclasses are subtypes. It is statically typed, via a type and effect system which turns out to be sound w.r.t. the operational semantics. Reclassification is achieved by dynamically changing the class membership of objects; correspondingly, the type system guarantees that objects will never access non-existing class components.

To develop the example in this section, we will refer to the second version of the language (known as $\mathcal{F}ickle_{II}$ [DDDG02]), even if a third one has improved it [DDG03], because we believe the former to be the main reference for that language.

In the `Fickle` scenario, an abstract class C has two non-overlapping concrete subclasses A and B, where the three classes must be of two different kinds: C is a *root* class, whereas A and B are *state* ones. In fact, one finds in root classes, such as C, the declaration of the (private) attributes (a.k.a. fields) and the (public) methods which are *common* to its state subclasses. On the other hand, state classes, such as A and B, are intended to serve as targets of reclassifications, hence their declaration contains the extra attributes and methods that *exclusively* belong to each of them.

The *reclassification* mechanism allows one object in a state class, say A, to become an object of the state class B (or, viceversa, moving from B to A) through the execution of a reclassification expression. The semantics of this operation, which may appear in the body of methods, is that the *attributes* of the object belonging to the source class are removed, those common to the two classes (which are in C) are retained, and the ones belonging to the target class are added to the object itself, *without* changing its identity. The same happens to the *methods* component, with the difference that the abstract methods declared in C (therefore common to A and B)

```
abstract root class Person extends Object {
   string name;

   abstract void employment(int n) {Person};
}

state class Student extends Person {
   int id;

   Student(string s, int m) { } { name := s; id := m };

   void employment(int n) {Person} { this⇓Worker; salary := n };
}

state class Worker extends Person {
   int salary;

   Worker(string s, int n) { } { name := s; salary := n };

   void employment(int n) { } { salary := salary + n };
   void registration(int m) {Person} { this⇓Student; id := m };
}
```

Figure 3 – Example in `Fickle`

may have different bodies in the two subclasses: when this is the case, reclassifying an object means replacing the bodies of the involved methods, too.

In the example of Figure 3, written in `Fickle` syntax, we first introduce the class `Person`, with an attribute to *name* a person and an abstract method to *employ* him/her. Then we add two subclasses, to model *students* and *workers*, with the following intended meaning. The `Student` class extends `Person` via a registration number (`id` attribute) and by instantiating the `employment` method. The `Worker` class extends `Person` via a remuneration information (`salary` attribute), a different `employment` method, and the extra `registration` method to register as a student (notice that, in our example, students and workers are mutually exclusive).

The root class `Person` defines the attributes and methods common to its state subclasses `Student` and `Worker` (notice that, being its `employment` method abstract, the root class itself must be abstract, therefore not supplying any constructor).

The classes `Student` and `Worker`, being subclasses of a root one (i.e. `Person`), must be state classes, which means that may be used as targets of reclassifications. Annotations, like `{ }` and `{Person}`, placed before the bodies of the methods, are named *effects* and are intended to list the root classes of the objects that may be reclassified by invoking those methods: in particular, the empty effect `{ }` cannot cause any reclassification and the non-empty effect `{Person}` allows to reclassify objects of

its subclasses. Let us now consider the following program fragment:

1. Person p, q;
2. p := **new** Student("Alice", 45);
3. q := **new** Worker("Bob", 27K);

After these lines, the variables `p` and `q` are bound to a `Student` and a `Worker` objects, respectively. To illustrate the key points of the reclassification mechanism, we make Bob become a student, and Alice first become a worker and then get a second job:

4. q.registration(57);
5. p.employment(30K);
6. p.employment(14K);

Code line 4, by sending the `registration` message to the object `q`, causes the execution of the reclassification *expression* `this`$\Downarrow$`Student`: before its execution, the receiver `q` is an object of the `Worker` class, therefore it contains the `salary` attribute; after it, `q` is *reclassified* into the `Student` class, hence `salary` is removed, `name` is not affected, and the `id` attribute is added and instantiated with the actual parameter.

Coming to the second object `p`, belonging to `Student` and representing Alice, line 5 carries out exactly the opposite operation w.r.t. line 4, by reclassifying `p` into the `Worker` class via the expression `this`$\Downarrow$`Worker`, with the result that `id` is no longer available, `name` preserves its value, and `salary` is added and instantiated.

The following line 6, therefore, selects the `employment` method from `Worker`, not from `Student` as before, because the object `p` has been reclassified in the meantime. This latter invocation of `employment` has the effect of augmenting Alice's income by the actual parameter value, thus allowing us to model a sort of multi-worker.

7.2 Desiderata

In this section, we devise the "ideal" behaviour of $\lambda\mathcal{O}bj+$ w.r.t. the reclassification goal, without requiring the typability of the terms that we introduce.

It is apparent that the main tool provided by our calculus to mimic `Fickle`'s reclassification mechanism is the self-extension primitive; precisely, we need a *reversible* extension functionality, to be used first to extend an object with new methods and later *to remove* from the resulting object some of its methods. Hence, an immediate solution would rely on a massive use of the self-extension primitive, e.g. as follows:

$$
\begin{array}{lcl}
\mathit{alice} & \triangleq & \langle\ \mathit{name} = \lambda s.\ \text{“}\mathit{Alice}''\text{,} \\
& & \quad \mathit{reg} = \lambda s, m.\ \langle\ \langle\ s \prec\!\!\leftarrow \mathit{id} = \lambda s'.\ m\ \rangle \\
& & \qquad\qquad\qquad \prec\!\!\leftarrow \mathit{emp} = \lambda s', n.\ s \Leftarrow \mathit{emp}(n)\ \rangle\text{,} \\
& & \quad \mathit{emp} = \lambda s, n.\ \langle\ \langle\ \langle\ s \prec\!\!\leftarrow \mathit{sal} = \lambda s'.\ n\ \rangle \\
& & \qquad\qquad\qquad\quad \prec\!\!\leftarrow \mathit{reg} = \lambda s', m.\ s \Leftarrow \mathit{reg}(m)\ \rangle \\
& & \qquad\qquad\qquad\quad \prec\!\!\leftarrow \mathit{emp} = \lambda s', n'.\ \langle\ s' \prec\!\!\leftarrow \mathit{sal} = \lambda s''.\ (s' \Leftarrow \mathit{sal}) + n'\ \rangle\ \rangle
\end{array}
$$

To model the example of Figure 3 in $\lambda\mathcal{O}bj+$, we have defined the *alice* term for representing Alice as a person. Now, it can be extended to either a student or a worker via the *reg* (i.e. registration) or *emp* (i.e. employment) methods, which are intended to play the role of the `Student` and `Worker` constructors of Section 7.1, respectively. We illustrate the behaviour of the former; in fact, *alice* becomes a student through the *reg* method, which adds *id* to the receiver and overrides the *emp* method:

$$
\begin{array}{lcl}
\mathit{alice} \Leftarrow \mathit{reg}(45) & \overset{ev}{\twoheadrightarrow} & \mathit{alice}_S \triangleq \langle\ \mathit{name} = \text{“}\mathit{Alice}''\text{,}\ \mathit{reg} = \ldots\text{,}\ \mathit{emp} = \ldots\text{,} \\
& & \qquad \mathit{id} = \lambda s.\ 45\text{,} \\
& & \qquad \mathit{emp} = \lambda s, n.\ \mathit{alice} \Leftarrow \mathit{emp}(n)\ \rangle
\end{array}
$$

In this way, the prototype *alice* is stored in the body of the novel *emp* method in the perspective of a reclassification (no matter if a cascade of *reg* is invoked and *emp* methods are stacked, because eventually the present version of *emp* is executed[2]).

Then, $alice_S$ can be reclassified into a worker via the invocation of such an *emp*, which sends to the original *alice* its *former* version (i.e. *alice*'s third method):

$$alice_S \Leftarrow emp(30K) \quad \overset{ev}{\twoheadrightarrow} \quad alice_W \quad \triangleq \quad \begin{array}{l} \langle\ name = \text{“}Alice''\text{”},\ reg = \ldots,\ emp = \ldots, \\ \quad sal = \lambda s.\ 30K, \\ \quad reg = \lambda s, m.\ alice \Leftarrow reg(m), \\ \quad emp = \lambda s, n.\ \langle\ s \prec\!\!\!- sal = \lambda s'.\ (s \Leftarrow sal) + n\ \rangle\ \rangle \end{array}$$

As the reader can see, the effect of this message is that the methods which characterize a student are removed (by coming back to *alice*) and those needed by a worker, in turn, extend *alice*; notice that the novel version of *emp* models the multi-worker.

To finalize Section's 7.1 example, $alice_W$'s income may be increased by means of a call to such a version of *emp*, which has overridden *alice*'s third method:

$$alice_W \Leftarrow emp(14K) \quad \overset{ev}{\twoheadrightarrow} \quad alice_{W_2} \quad \triangleq \quad \begin{array}{l} \langle\ name = \text{“}Alice''\text{”},\ reg = \ldots,\ emp = \ldots, \\ \quad sal = \ldots,\ reg = \ldots,\ emp = \ldots, \\ \quad sal = \lambda s.\ (alice_W \Leftarrow sal) + 14K\ \rangle \end{array}$$

Tipability. The encoding devised in this section may be seen as a reasonable solution to emulate object reclassification in $\lambda\mathcal{O}bj+$; however, the actual free use of the self-extension primitive does not allow us to type the terms introduced.

The point is that the `self` variable (named s, s', s'' to save space) cannot be used in the body of a method added by self-extension to remove methods, in the attempt to restore the receiver before its extension (it is the case of *emp*'s body, added by the second method *reg* and, symmetrically, *reg*'s body, added by *emp*).

We can discuss the issue via the minimal (hence simpler than *alice*) term:

$$r \quad \triangleq \quad \langle\ extend = \lambda s.\ \langle\ s \prec\!\!\!- delete = \lambda s'.\ s\ \rangle\ \rangle \tag{29}$$

where the difficulty concerns the type returned by the *delete* method:

$$r\ :\ \mathtt{pro}\, t.\langle\!\langle\ extend : t \triangleleft \langle delete \rangle,\ delete : ?\ \rangle\!\rangle \triangleleft \langle extend \rangle$$

We first observe that the type variable t would not be a suitable candidate for *delete*, because t, within the scope of the above `pro` binder, is intended to represent the receiver, i.e. in the *delete* case at hand, the object *already* extended and therefore containing the *delete* method.

A second attempt would be typing r itself with the type returned by *delete*:

$$r\ :\ R \quad \triangleq \quad \mathtt{pro}\, t.\langle\!\langle\ extend : t \triangleleft \langle delete \rangle,\ delete : R\ \rangle\!\rangle \triangleleft \langle extend \rangle$$

That is, the correct type R should satisfy a recursion equation. However, $\lambda\mathcal{O}bj+$'s recursion mechanisms are not powerful enough to express such a type, hence we are devoting the remaining part of Section 7 to design alternative and typable encodings.

[2] An alternative solution is that *reg* in *alice* overrides itself as $reg = \lambda s', m'.\ \langle\ s' \prec\!\!\!- id = \lambda s''.\ m'\ \rangle$; in such an equivalent case only *id* methods would be stacked, rather than $\langle id, emp \rangle$ pairs.

7.3 The runtime solution

A first possibility to circumvent the tipability problem arised in Section 7.2 is very plain: at first we extend an object with new methods, and from then we keep just overriding the resulting object, without removing methods from it. That is, the first use of the self-extension leads to object extension, whereas all the following ones to object override. We may then model the reclassification in $\lambda\mathcal{O}bj+$ as follows:

$$\begin{array}{lll} alice' & \triangleq & \langle\ name = \lambda s.\ \text{“Alice”},\\ & & \quad reg = \lambda s, m.\ \langle\ \langle\ s \prec\!\!\leftarrow id = \lambda s'.\ m\ \rangle\\ & & \qquad\qquad \prec\!\!\leftarrow sal = \lambda s'.\ 0\ \rangle,\\ & & \quad emp = \lambda s, n.\ \langle\ \langle\ \langle\ s \prec\!\!\leftarrow id = \lambda s'.\ 0\ \rangle\\ & & \qquad\qquad \prec\!\!\leftarrow sal = \lambda s'.\ n\ \rangle\\ & & \qquad\qquad \prec\!\!\leftarrow emp = \lambda s', n'.\ \langle\ \langle\ s' \prec\!\!\leftarrow id = \lambda s''.\ 0\ \rangle\\ & & \qquad\qquad\qquad\qquad \prec\!\!\leftarrow sal = \lambda s''.\ (s' \Leftarrow sal) + n'\ \rangle\ \rangle\ \rangle \end{array}$$

As the reader can inspect, in this alternative Alice's encoding the variables representing `self` (again named s, s', s'') are never used in a method body to represent an object *without* the method being defined. This crucial fact holds also for the rightmost *sal*, where s' refers to an object where that method is *already* available; such a property can be checked *syntactically*, hence *alice*′ may be given the following type:

$$\begin{array}{lll} alice' & : & \mathtt{pro}\, t.\langlle\ name : String,\\ & & \qquad reg : \mathbb{N} \to (t \triangleleft \langle id, sal\rangle),\ emp : \mathbb{N} \to (t \triangleleft \langle id, sal\rangle),\\ & & \qquad id : \mathbb{N},\ sal : \mathbb{N}\\ & & \quad \ranglerle \triangleleft \langle name,\ reg,\ emp\rangle \end{array} \tag{30}$$

The price to pay for typability is that the objects playing the roles of students and workers will contain more methods than needed (*all* the methods involved), because no method can be removed. In the present example, when *alice*' registers as a student, *id* and *sal* are added permanently to the interface:

$$alice' \Leftarrow reg(45) \quad \overset{ev}{\rightarrow} \quad alice'_S \quad \triangleq \quad \langle\ name = \ldots,\ reg = \ldots,\ emp = \ldots,\ id = \lambda s.\ 45,\ sal = \lambda s.\ 0\ \rangle$$

Therefore, the type system will not detect type errors related to uncorrect method calls. In fact, $alice'_S$ is intended to represent a student, but in practice we have to distinguish between students and workers via the runtime answers to the *id* and *sal* (representing students' and workers' attributes, respectively) method invocations: non-zero values (such as 45, returned by *id*) are informative of genuine attributes, while zero values (returned by *sal*) tell us that the corresponding attribute is not significant. This solution is reminiscent of an approach to reclassification via *wide classes*, requiring runtime tests to diagnose the presence of fields [Ser99].

The next step of our example is the reclassification of Alice into a worker[3]:

$$\begin{array}{lll} alice'_S \Leftarrow emp(30K) \quad \overset{ev}{\rightarrow} \quad alice'_W & \triangleq & \langle\ name = \ldots,\ reg = \ldots,\\ & & \quad id = \lambda s.\ 0,\ sal = \lambda s.\ 30K,\\ & & \quad emp = \lambda s, n.\ \langle\ \langle\ s \prec\!\!\leftarrow id = \lambda s'.\ 0\ \rangle\\ & & \qquad\qquad \prec\!\!\leftarrow sal = \lambda s'.\ (s \Leftarrow sal) + n\ \rangle\ \rangle \end{array}$$

The consequence of this call to (the original) *emp* is that *id* and *sal* swap their role, thus making effective the reclassification, and a new version of *emp* itself is embedded

[3]Notice that, to ease readibility, we will omit from now on the overriden methods, if the latter have definitively become garbage (in the case: the inner versions of *emp*, *id*, *sal*).

in the interface. We observe that such a novel *emp* (which increments the salary *sal*) works correctly not only with the usual multi-worker operation:

$$alice'_W \Leftarrow emp(14K) \quad \overset{ev}{\rightarrow} \quad alice'_{W_2} \quad \triangleq \quad \langle\, name = \ldots,\ reg = \ldots,\ emp = \ldots,\ id = \lambda s.\ 0,\ sal = \lambda s.\ (alice'_W \Leftarrow sal) + 14K \,\rangle$$

but also in the case of a further reclassification of $alice'_W$ into a student, because setting ex-novo a salary is equivalent to adding it to the zero value stored by *reg*.

Finally, a couple of remarks about the relationship of the two *emp* versions with the type (30). First, the fact that the overridden *emp* (i.e. the one belonging to $alice'$) extends the receiver via *id* and *sal* but overrides itself is clearly expressed by its type $\mathbb{N} \to (t \triangleleft \langle id, sal \rangle)$. Second, the redundant *id* version contained by the overriding *emp* (i.e. the one that appears in $alice'_W$) is hence necessary to respect such a type.

7.4 Creating new objects

A way to achieve the possibility to remove methods from an object is by creating new objects. To illustrate such an insight, we pick out the following example:

$$r' \quad \triangleq \quad \langle\, extend = \lambda s.\ \langle\, extend = \lambda s'.\ s',\ delete = \lambda s'.\ s \rangle\, \rangle$$

which models the same behavior of the r object (29), introduced in Section 7.2 to enlighten the typability problem that we want to encompass. In the present case, the method *delete* is allowed by the type system to return its prototype object (represented by the `self` variable s), because such a method belongs to a completely *new object*, not to an object which has *extended* its prototype (as it was in Section 7.2):

$$r' \ :\ R' \quad \triangleq \quad \mathtt{pro}\, t.\langle\!\langle\, extend : \mathtt{pro}\, t'.\ \langle\!\langle\, extend : t',\ delete : t \,\rangle\!\rangle \triangleleft \langle extend,\ delete \rangle \,\rangle\!\rangle \triangleleft \langle extend \rangle$$

The reader may observe how the type R' reflects the explanation given above: a new object is generated via the *extend* method and represented by t'; within such an object, the *delete* method refers to the prototype object, represented by t.

We apply the idea to our working example; combining the self-extension primitive with the generation of new objects leads to a third Alice's representation:

$$\begin{aligned} alice'' \quad \triangleq \quad & \langle\, name = \lambda s.\ \text{“Alice”}, \\ & \quad reg = \lambda s, m.\ \langle\, name = \lambda s'.\ s \Leftarrow name, \\ & \qquad\qquad id = \lambda s'.\ m, \\ & \qquad\qquad emp = \lambda s', n.\ \langle\, s \prec\!\!\leftarrow sal = \lambda s''.\ 0 \,\rangle \Leftarrow emp(n) \,\rangle, \\ & \quad emp = \lambda s, n.\ \langle\, \langle\, s \prec\!\!\leftarrow sal = \lambda s'.\ n \,\rangle \\ & \qquad\qquad \prec\!\!\leftarrow emp = \lambda s', n'.\ \langle\, s' \prec\!\!\leftarrow sal = \lambda s''.\ (s' \Leftarrow sal) + n' \,\rangle \,\rangle \,\rangle \end{aligned}$$

The novelty of the present solution amounts to the fact that the *reg* method creates *a new object* from scratch, equipped with three methods: the first one copies the *name* value from its prototype, the second method sets the *id* attribute, and, the key point, the *emp* method is allowed to refer back to the prototype object to prepare for a potential worker reclassification. As argued above, this latter method is typable, conversely to its version in *alice* (Section 7.2), because it is not added by self-extension, but it belongs to a different object, created ex-novo. In the end, the current Alice's representation is type-checked as follows (note that its third method *emp* is not

problematic, being simpler than in previous Section 7.3):

$$
\begin{array}{lcl}
alice'' & : & T \\
T & \triangleq & \mathtt{pro}\, t.\langle\!\langle\ name : String, \\
 & & \quad reg : \mathbb{N} \to \mathtt{pro}\, t'.\ \langle\!\langle\ name : String, \\
 & & \qquad\qquad id : \mathbb{N}, \\
 & & \qquad\qquad emp : \mathbb{N} \to (t \triangleleft \langle sal \rangle) \\
 & & \qquad \rangle\!\rangle \triangleleft \langle name,\ id,\ emp \rangle, \\
 & & \quad emp : \mathbb{N} \to (t \triangleleft \langle sal \rangle), \\
 & & \quad sal : \mathbb{N} \\
 & & \rangle\!\rangle \triangleleft \langle name,\ reg,\ emp \rangle
\end{array}
$$

where it is apparent that both the *emp* versions add *sal* to *alice″*'s interface.

Then, the outcome of Alice's registration is the following:

$$
\begin{array}{lcl}
alice'' \Leftarrow reg(45) & \overset{ev}{\to} & alice''_S \triangleq \langle\ name = \lambda s.\ alice'' \Leftarrow name, \\
 & & \quad id = \lambda s.\ 45, \\
 & & \quad emp = \lambda s, n.\ \langle\ alice'' \prec\!\!\leftarrow sal = \lambda s'.\ 0\ \rangle \Leftarrow emp(n)\ \rangle
\end{array}
$$

$$
\begin{array}{lcl}
alice''_S & : & \mathtt{pro}\, t''.\ \langle\!\langle\ name : String, \\
 & & \quad id : \mathbb{N}, \\
 & & \quad emp : \mathbb{N} \to (T \triangleleft \langle sal \rangle) \\
 & & \rangle\!\rangle \triangleleft \langle name,\ id,\ emp \rangle
\end{array}
$$

One can see in this latter type that, coherently, the *emp* method adds *sal* to the prototype *alice″*. We observe also that, in *emp*'s body, a "local" version of *sal* is added on the fly to the receiver (*alice″*, in the case) before the call to the outer *emp*. This is necessary to guarantee the correctness of the protocol in the event of a call to *alice″*'s *emp* before than *reg* (an example that we do not detail here): *emp* overrides itself, thus losing from then the possibility to set the salary from scratch (see the *alice″* term), which must be hence incremented starting from zero.

The chance to send *emp* to the prototype *alice″* is crucial for the reclassification:

$$
\begin{array}{lcl}
alice''_S \Leftarrow emp(30K) & \overset{ev}{\to} & alice''_W \triangleq \langle\ name = \ldots,\ reg = \ldots, \\
 & & \quad sal = \lambda s.\ 30K, \\
 & & \quad emp = \lambda s, n.\ \langle\ s \prec\!\!\leftarrow sal = \lambda s'.\ (s \Leftarrow sal) + n\ \rangle\ \rangle
\end{array}
$$

$$
\begin{array}{lcl}
alice''_W & : & \mathtt{pro}\, t.\langle\!\langle\ name : String, \\
 & & \quad reg : \mathbb{N} \to \mathtt{pro}\, t'.\ \langle\!\langle\ name : String, \\
 & & \qquad\qquad id : \mathbb{N}, \\
 & & \qquad\qquad emp : \mathbb{N} \to t \\
 & & \qquad \rangle\!\rangle \triangleleft \langle name,\ id,\ emp \rangle, \\
 & & \quad sal : \mathbb{N}, \\
 & & \quad emp : \mathbb{N} \to t \\
 & & \rangle\!\rangle \triangleleft \langle name,\ reg,\ sal,\ emp \rangle
\end{array}
$$

where the presence of the salary in the new interface is reflected by both *emp*'s types.

We end by adding the usual second job to Alice (the type is the same of $alice''_W$):

$$
\begin{array}{lcl}
alice''_W \Leftarrow emp(14K) & \overset{ev}{\to} & alice''_{W_2} \triangleq \langle\ name = \ldots,\ reg = \ldots,\ emp = \ldots, \\
 & & \quad sal = \lambda s.\ (alice''_W \Leftarrow sal) + 14K\ \rangle
\end{array}
$$

Discussion. It is apparent that the opposite reclassification direction (Alice first becoming a worker and then a student) would produce terms behaviourally equivalent to $alice''_W$ and $alice''_S$, even though not syntactically identical.

We remark also that in fact a couple of choices is already feasible, if one decides to combine self-extensions and new objects: in principle, there is no reason to prefer the encoding that we have illustrated to the symmetrical one (simpler, in the case), where students are modeled via self-extensions and workers through new objects.

To conclude, the reader might wonder about the asymmetry of the solution developed in this section, as students are managed via new objects and workers through self-extensions. Actually, in Section 7.2 we have shown that modeling the reclassification by means of the sole self-extension mechanism leads to non-typable terms. On the opposite side, it is always possible to encode the reclassification via only new objects (to manage also workers), without the need of the self-extension:

$$
\begin{array}{lcl}
alice''' & \triangleq & \langle\ name = \lambda s.\ \text{“Alice”},\\
 & & \quad reg = \lambda s, m.\ \langle\ name = \lambda s'.\ s \Leftarrow name,\\
 & & \qquad\qquad id = \lambda s'.\ m,\\
 & & \qquad\qquad emp = \lambda s', n.\ s \Leftarrow emp(n)\ \rangle,\\
 & & \quad emp = \lambda s, n.\ \langle\ name = \lambda s'.\ s \Leftarrow name,\\
 & & \qquad\qquad sal = \lambda s'.\ n,\\
 & & \qquad\qquad emp = \lambda s', n'.\ \langle\ s' \prec\!\!\leftarrow sal = \lambda s''.\ (s' \Leftarrow sal) + n'\ \rangle,\\
 & & \qquad\qquad reg = \lambda s', m.\ s \Leftarrow reg(m)\ \rangle\ \rangle
\end{array}
$$

However, in this section we have tried to push the self-extension, which is the technical novelty of this paper, to its limit (i.e. typability). We believe that such an effort is interesting per se; moreover, the “mixed” solution which arises from our investigation leads to a more compact encoding, giving the benefit of code reuse.

8 Related work

With an aim similar to that of our work, several efforts have been carried out in recent years for the sake of providing static type systems for object-oriented languages that change, at runtime, the behaviour of objects. In this section, first we discuss the approaches in the literature by considering separately the two main categories of prototype-based and class-based languages, afterwards we survey the relationship between object extension and object subsumption.

8.1 Prototype-based scenario

A few works consider the problem of defining static type disciplines for `JavaScript`, a prototype-based, dynamically typed language where objects can be modified at runtime and errors caused by calls to undefined methods may occur.

Zhao, in [Zha10, Zha12], presents a static type inference algorithm for a fragment of `JavaScript` and suggests two type disciplines for preventing undefined method calls. Similarly to the $\lambda\mathcal{O}bj+$ calculus, `JavaScript` allows self-inflicted extensions; to deal with this feature, some ideas shared with our approach are adopted:

- the distinction between `pro`-types and `obj`-types;
- the distinction between “available” and “reserved” methods;
- the mechanisms to mark the migration of a method from reserved to available.

On the other hand, the main differences or extra features are the following:

- `JavaScript` allows strong update, i.e. overriding a method with a different type: the type system accommodates, in a limited way, this functionality;
- the types are defined by means of a set of subtyping constraints;
- the syntax is completely different.

A static type system for quite a rich subset of `JavaScript` is also proposed by Chugh and co-workers in [CHJ12]. The considered features are imperative updates (i.e. updates that change the set of methods of an object by adding and also subtracting methods) and arrays, which in `JavaScript` can be homogeneous (when all the elements have the same type) but also heterogeneous, like tuples. As the syntax makes no distinction between these two kinds of arrays, to form the correct type can be challenging. In order to deal with subtyping and inheritance, the article contains a further elaboration of our idea of splitting the list of methods into reserved and available parts.

Vouillon presents in [Vou01] a prototype-based calculus containing the "view" mechanism, which allows to change the interface between an object and the environment, thus permitting an object to hide part of its methods in some context. Similarly to our work, the author defines a distinction between `pro`-types and `obj`-types.

8.2 Class-based scenario

The typical setting where class-based languages are investigated is a Java-like environment. In the previous Section 7 we have considered object *reclassification*, a feature introduced in the class-based paradigm, and we have experimented with modeling in $\lambda\mathcal{O}bj+$ the reclassification mechanisms implemented in `Fickle` [DDDG02].

We complete now the survey of the involved related work by presenting more recent contributions that fall in the same class-based category.

Object evolution. Actually, Cohen and Gil's work [CG09], about the introduction of *object evolution* into statically typed languages, is much related to reclassification, because evolution is a restriction of reclassification, by which objects may only gain, but never lose their capabilities (hence it may be promptly mimicked in $\lambda\mathcal{O}bj+$).

An evolution operation (which may be of three non-mutually exclusive variants, based respectively on inheritance, mixins, and shakeins) takes at runtime an instance of one class and replaces it with an instance of a selected subclass. The monotonicity property granted by such a kind of dynamic change makes easier to maintain static type-safety than in general reclassification.

In the end, the authors experiment with an implementation of evolution in `Java`, based on the idea of using a forward pointer to a new memory address to support the objects which have evolved, starting from the original non-evolved object.

Context and roles. *Role-based* languages supply role modeling (via a context declaration, which is a set of roles that represent collaboration carried out in that context, e.g. between an employer and its employees) and object adaptation (that is, a dynamic change of role, to partecipate in a context by assuming one of its roles). These mechanisms are implemented in the Java-based `EpsilonJ` language. However, dynamically acquired methods obtained by assuming roles have to be invoked by means of *down-casting*, which is a well-known type unsafe operation.

Therefore, Kamina and Tamai introduce in [KT10] an extension of `Java`, named `NextEJ`, to combine the object-based adaptation mechanisms of `EpsilonJ` and the object-role binding provided by *context-oriented* languages.

In fact, the authors model in `NextEJ` the *context activation scope*, adopted from the latter languages, and prove that such a mechanism is type sound by using a small calculus which formalizes the core features of `NextEJ`.

Talents. The goal of Ressia and co-workers [RGN+11], who settle in an untyped scenario, is to manage the reuse of methods at runtime in a way that supports behavioural composition. To pursue this aim, they introduce *talents* as sets of methods which can be assigned to any object to add, remove or update methods in its interface.

Technically, within a chosen language-independent metamodel for software analysis, a talent is an object belonging to a standard class, named `Talent`, which can be acquired (via a suitable `acquire` primitive) by any object, which is then adapted. The crucial operational characteristics of talents are that they are scoped dynamically and that their composition order is irrelevant. However, when two talents with different implementations of the same method are composed a conflict arises, which has to be resolved either through aliasing (the name of the method in a talent is changed) or via exclusion (the method is removed from a talent before composition).

The authors have experimented with implementing talents in the open-source `Pharo Smalltalk` system. Later, in [RGN+14], they discuss the possibility of porting talents to a statically typed language, such as `Java`, without weakening the existing type system, which would require that methods that replace existing ones do not alter their signatures.

8.3 Object extension vs. subsumption

Several calculi proposed in the literature combine object extension with object subsumption ([RS98, FM95, Liq97, BBDL97, Rém98]).

In [RS98], it is presented a calculus where it is possible to first subsume (forget) an object component, and then re-add it again with a type which may be incompatible with the forgotten one. In order to guarantee the soundness of the type system, *method dictionaries* are used inside objects, which link correctly method names and method bodies.

Approaches to subsumption similar to the one presented in this work can be found in [FM95, Liq97, BBDL97, Rém98]. In [Liq97], an extension of Abadi and Cardelli's Object Calculus is presented; roughly speaking, we can say that `pro`-types and `obj`-types in the present article correspond to "diamond-types" and "saturated-types" in that work. Similar ideas can be found in [Rém98], although the type system there presented permits also a form of self-inflicted extension. However, in that type system, a method `m` performing a self-inflicted extension needs to return a rigid object whose type is fixed in the declaration of the body of `m`. As a consequence, the following expressions are not typable in that system:

$$\langle\langle \texttt{point} \prec\!\!\!\!- \texttt{newm} = \ldots\rangle \Leftarrow \texttt{add\_set\_col}\rangle \Leftarrow \texttt{newm}$$

$$\langle\langle \texttt{point} \Leftarrow \texttt{add\_set\_col}\langle \prec\!\!\!\!- \texttt{newm} = \ldots\rangle$$

Another type system for the $\lambda\mathcal{O}bj$ calculus is presented in [BBDL97]; such a type system uses a refined notion of subtyping that allows to type also *binary methods*.

A The Type Assignment Rules

Well-formed Contexts

$$\frac{}{\varepsilon \vdash ok}\ (Cont{-}\varepsilon)$$

$$\frac{\Gamma \vdash \tau : T \qquad x \notin dom(\Gamma)}{\Gamma, x : \tau \vdash ok}\ (Cont{-}x)$$

$$\frac{\Gamma \vdash \rho : T \qquad t \notin dom(\Gamma) \qquad \rho \text{ a } \texttt{pro}\text{-type}}{\Gamma, t <\!\# \rho \vdash ok}\ (Cont{-}t)$$

Well-formed Types

$$\frac{\Gamma \vdash ok}{\Gamma \vdash \texttt{pro}\, t.\langle\!\langle \epsilon \rangle\!\rangle : T}\ (Type{-}Pro_{\langle\!\langle\,\rangle\!\rangle})$$

$$\frac{\Gamma, t <\!\# \texttt{pro}\, t.\langle\!\langle R \rangle\!\rangle \vdash \sigma : T \qquad \mathtt{m} \notin \mathcal{M}(R)}{\Gamma \vdash \texttt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma \rangle\!\rangle : T}\ (Type{-}Pro)$$

$$\frac{\Gamma \vdash \rho <\!\# \texttt{pro}\, t.\langle\!\langle R \rangle\!\rangle \qquad \{M\} \subseteq \mathcal{M}(R)}{\Gamma \vdash \rho \triangleleft \langle M \rangle : T}\ (Type{-}Extend)$$

$$\frac{\Gamma \vdash \tau : T \qquad \Gamma \vdash \upsilon : T}{\Gamma \vdash \tau \to \upsilon : T}\ (Type{-}Arrow)$$

$$\frac{\Gamma \vdash ok}{\Gamma \vdash \iota : T}\ (Type{-}Const)$$

Matching Rules

$$\frac{\Gamma \vdash \texttt{pro}\, t.\langle\!\langle R' \rangle\!\rangle \triangleleft \langle M' \rangle : T \qquad \Gamma \vdash \texttt{pro}\, t.\langle\!\langle R \rangle\!\rangle \triangleleft \langle M \rangle : T \qquad R \subseteq R' \qquad \{M\} \subseteq \{M'\}}{\Gamma \vdash \texttt{pro}\, t.\langle\!\langle R' \rangle\!\rangle \triangleleft \langle M' \rangle <\!\# \texttt{pro}\, t.\langle\!\langle R \rangle\!\rangle \triangleleft \langle M \rangle}\ (Match{-}Pro)$$

$$\frac{\Gamma, t <\!\# \rho, \Gamma' \vdash \rho \triangleleft \langle M \rangle <\!\# \xi}{\Gamma, t <\!\# \rho, \Gamma' \vdash t \triangleleft \langle M \rangle <\!\# \xi}\ (Match{-}Var)$$

$$\frac{\Gamma \vdash t \triangleleft \langle M' \rangle : T \qquad \{M\} \subseteq \{M'\}}{\Gamma \vdash t \triangleleft \langle M' \rangle <\!\# t \triangleleft \langle M \rangle}\ (Match{-}t)$$

Type Rules for Lambda Terms

$$\frac{\Gamma \vdash ok}{\Gamma \vdash c : \iota}\ (Const)$$

$$\frac{\Gamma, x : \tau, \Gamma' \vdash ok}{\Gamma, x : \tau, \Gamma' \vdash x : \tau} \ (Var)$$

$$\frac{\Gamma, x : \tau \vdash e : \upsilon}{\Gamma \vdash \lambda x.e : \tau \to \upsilon} \ (Abs)$$

$$\frac{\Gamma \vdash e_1 : \tau \to \upsilon \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\, e_2 : \upsilon} \ (Appl)$$

Type Rules for Object Terms

$$\frac{\Gamma \vdash ok}{\Gamma \vdash \langle\,\rangle : \mathtt{pro}\, t.\langle\!\langle \epsilon \rangle\!\rangle} \ (Empty)$$

$$\frac{\Gamma \vdash e : \mathtt{pro}\, t.\langle\!\langle R \rangle\!\rangle \triangleleft \langle M \rangle \qquad \Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R, R' \rangle\!\rangle \triangleleft \langle M \rangle : T}{\Gamma \vdash e : \mathtt{pro}\, t.\langle\!\langle R, R' \rangle\!\rangle \triangleleft \langle M \rangle} \ (Pre{-}Extend)$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \rho \\ \Gamma \vdash \rho <\!\# \,\mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma \rangle\!\rangle \triangleleft \langle M \rangle \\ \Gamma, t <\!\# \,\mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma \rangle\!\rangle \triangleleft \langle M, \mathtt{m} \rangle \vdash e_2 : t \to \sigma\end{array}}{\Gamma \vdash \langle e_1 \prec\!\!- \mathtt{m} = e_2 \rangle : \rho \triangleleft \mathtt{m}} \ (Extend)$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \rho \\ \Gamma \vdash \rho <\!\# \,\mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma \rangle\!\rangle \triangleleft \langle M, \mathtt{m} \rangle \\ \Gamma, t <\!\# \,\mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma \rangle\!\rangle \triangleleft \langle M, \mathtt{m} \rangle \vdash e_2 : t \to \sigma\end{array}}{\Gamma \vdash \langle e_1 \prec\!\!- \mathtt{m} = e_2 \rangle : \rho} \ (Override)$$

$$\frac{\Gamma \vdash e : \rho \qquad \Gamma \vdash \rho <\!\# \,\mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma \rangle\!\rangle \triangleleft \langle M, \mathtt{m} \rangle}{\Gamma \vdash e \Leftarrow \mathtt{m} : \sigma[\rho/t]} \ (Send)$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \rho \\ \Gamma \vdash \rho <\!\# \,\mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma \rangle\!\rangle \triangleleft \langle M, \mathtt{m} \rangle \\ \Gamma, t <\!\# \,\mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma \rangle\!\rangle \triangleleft \langle M, \mathtt{m} \rangle \vdash e_2 : t \to (t \triangleleft \langle N \rangle)\end{array}}{\Gamma \vdash Sel(e_1, \mathtt{m}, e_2) : \sigma[(\rho \triangleleft \langle N \rangle)/t]} \ (Select)$$

B The Extra Rules for Subsumption

Extra Well-formed Types

$$\frac{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R \rangle\!\rangle \triangleleft \langle M \rangle : T}{\Gamma \vdash \mathtt{obj}\, t.\langle\!\langle R \rangle\!\rangle \triangleleft \langle M \rangle : T} \ (Type{-}Obj)$$

$$\frac{\Gamma \vdash \tau <\!\# \,\mathtt{obj}\, t.\langle\!\langle R \rangle\!\rangle \qquad \{M\} \subseteq \mathcal{M}(R)}{\Gamma \vdash \tau \triangleleft \langle M \rangle : T} \ (New - Type{-}Extend)$$

Extra Matching Rules

$$\frac{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R' \rangle\!\rangle \triangleleft \langle M' \rangle : T \qquad \Gamma \vdash \mathtt{obj}\, t.\langle\!\langle R \rangle\!\rangle \triangleleft \langle M \rangle : T \qquad R \subseteq R' \qquad \{M\} \subseteq \{M'\}}{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R' \rangle\!\rangle \triangleleft \langle M' \rangle \lessdot\!\!\# \mathtt{obj}\, t.\langle\!\langle R \rangle\!\rangle \triangleleft \langle M \rangle} \ (Promote)$$

$$\frac{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R' \rangle\!\rangle \triangleleft \langle M' \rangle : T \qquad \Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R \rangle\!\rangle \triangleleft \langle M \rangle : T \qquad R \subseteq R' \qquad \{M\} \subseteq \{M'\}}{\Gamma \vdash \mathtt{obj}\, t.\langle\!\langle R' \rangle\!\rangle \triangleleft \langle M' \rangle \lessdot\!\!\# \mathtt{obj}\, t.\langle\!\langle R \rangle\!\rangle \triangleleft \langle M \rangle} \ (Match{-}Obj)$$

$$\frac{\Gamma \vdash \upsilon' \lessdot\!\!\# \upsilon \qquad \Gamma \vdash \tau \lessdot\!\!\# \tau' \qquad \Gamma \vdash \upsilon : Rdg}{\Gamma \vdash \upsilon \to \tau \lessdot\!\!\# \upsilon' \to \tau'} \ (Match{-}Arrow)$$

New Type Rules for Expressions

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash \tau \lessdot\!\!\# \upsilon \qquad \Gamma \vdash \upsilon : Rgd}{\Gamma \vdash e : \upsilon} \ (Subsume)$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \rho \\ \Gamma \vdash \rho \lessdot\!\!\# \mathtt{obj}\, t.\langle\!\langle R, \mathtt{m} : \sigma \rangle\!\rangle \triangleleft \langle M \rangle \\ \Gamma, t \lessdot\!\!\# \mathtt{obj}\, t.\langle\!\langle R, \mathtt{m} : \sigma \rangle\!\rangle \triangleleft \langle M, \mathtt{m} \rangle \vdash e_2 : t \to \sigma\end{array}}{\Gamma \vdash \langle e_1 \prec\!\!\!\leftarrow \mathtt{m} = e_2 \rangle : \rho \triangleleft \mathtt{m}} \ (New{-}Extend)$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \rho \\ \Gamma \vdash \rho \lessdot\!\!\# \mathtt{obj}\, t.\langle\!\langle R, \mathtt{m} : \sigma \rangle\!\rangle \triangleleft \langle M, \mathtt{m} \rangle \\ \Gamma, t \lessdot\!\!\# \mathtt{obj}\, t.\langle\!\langle R, \mathtt{m} : \sigma \rangle\!\rangle \triangleleft \langle M, \mathtt{m} \rangle \vdash e_2 : t \to \sigma\end{array}}{\Gamma \vdash \langle e_1 \prec\!\!\!\leftarrow \mathtt{m} = e_2 \rangle : \rho} \ (New{-}Override)$$

$$\frac{\Gamma \vdash e : \rho \qquad \Gamma \vdash \rho \lessdot\!\!\# \mathtt{obj}\, t.\langle\!\langle R, \mathtt{m} : \sigma \rangle\!\rangle \triangleleft \langle M, \mathtt{m} \rangle}{\Gamma \vdash e \Leftarrow \mathtt{m} : \sigma[\rho/t]} \ (New{-}Send)$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \rho \\ \Gamma \vdash \rho \lessdot\!\!\# \mathtt{obj}\, t.\langle\!\langle R, \mathtt{m} : \sigma \rangle\!\rangle \triangleleft \langle M, \mathtt{m} \rangle \\ \Gamma, t \lessdot\!\!\# \mathtt{obj}\, t.\langle\!\langle R, \mathtt{m} : \sigma \rangle\!\rangle \triangleleft \langle M, \mathtt{m} \rangle \vdash e_2 : t \to (t \triangleleft \langle N \rangle)\end{array}}{\Gamma \vdash Sel(e_1, \mathtt{m}, e_2) : \sigma[(\rho \triangleleft \langle N \rangle)/t]} \ (New{-}Select)$$

Rules for Fixed Types

$$\frac{\Gamma \vdash ok}{\Gamma \vdash \iota : Rgd} \ (Rgd{-}Const)$$

$$\frac{\Gamma \vdash \mathtt{obj}\, t.\langle\!\langle R \rangle\!\rangle \triangleleft \langle M \rangle : T \qquad t \text{ covariant in } R \qquad \Gamma \vdash R : Rgd}{\Gamma \vdash \mathtt{obj}\, t.\langle\!\langle R \rangle\!\rangle \triangleleft \langle M \rangle : Rgd} \ (Rgd{-}Obj)$$

$$\frac{\Gamma, t \lessdot\!\!\# \mathtt{obj}\, t.\langle\!\langle R \rangle\!\rangle \triangleleft \langle M \rangle, \Gamma' \vdash t \triangleleft \langle N \rangle : T \qquad t \text{ covariant in } R}{\Gamma, t \lessdot\!\!\# \mathtt{obj}\, t.\langle\!\langle R \rangle\!\rangle \triangleleft \langle M \rangle, \Gamma' \vdash t \triangleleft \langle N \rangle : Rgd} \ (Rgd{-}Var)$$

$$\frac{\Gamma \vdash \upsilon : Rgd}{\Gamma \vdash \tau \to \upsilon : Rgd}\ (Rgd{-}Arrow)$$

References

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. 1996.

[BB99] Viviana Bono and Michele Bugliesi. Matching for the lambda calculus of objects. *Theoretical Computer Science*, 212(1-2):101–140, 1999.

[BBDL97] V. Bono, M. Bugliesi, M. Dezani-Ciancaglini, and L. Liquori. Subtyping Constraint for Incomplete Objects. In *Proc. of TAPSOFT/CAAP*, volume 1214, pages 465–477, 1997.

[BCC+96] K. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G. Leavens, and B. Pierce. On Binary Methods. *Theory and Practice of Object Systems*, 1(3), 1996.

[BF98] Viviana Bono and Kathleen Fisher. An imperative, first-order calculus with object extension. In Eric Jul, editor, *ECOOP'98 - Object-Oriented Programming, 12th European Conference, Brussels, Belgium, July 20-24, 1998, Proceedings*, volume 1445 of *Lecture Notes in Computer Science*, pages 462–497. Springer, 1998. URL: `https://doi.org/10.1007/BFb0054104`, `doi:10.1007/BFb0054104`.

[BL95] V. Bono and L. Liquori. A Subtyping for the Fisher-Honsell-Mitchell Lambda Calculus of Objects. In *Proc. of CSL*, volume 933, pages 16–30, 1995.

[BPF97] K. Bruce, L. Petersen, and A. Fiech. Subtyping Is Not a Good "Match" for Object-Oriented Languages. In *Proc. of ECOOP*, volume 1241, pages 104–127, 1997.

[Bru94] K.B. Bruce. A Paradigmatic Object–Oriented Programming Language: Design, Static Typing and Semantics. *Journal of Functional Programming*, 4(2):127–206, 1994.

[Car95] L. Cardelli. A Language with Distributed Scope. *Computing System*, 8(1):27–59, 1995.

[Cas95] G. Castagna. Covariance and contravariance: conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, 1995.

[Cas96] G. Castagna. *Object-Oriented Programming: A Unified Foundation*. Progress in Theoretical Computer Science. Birkäuser, Boston, 1996.

[CG09] Tal Cohen and Joseph Gil. Three approaches to object evolution. In Ben Stephenson and Christian W. Probst, editors, *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java, PPPJ 2009, Calgary, Alberta, Canada, August 27-28, 2009*, pages 57–66. ACM, 2009. URL: `http://doi.acm.org/10.1145/1596655.1596665`, `doi:10.1145/1596655.1596665`.

[CHJ12] Ravi Chugh, David Herman, and Ranjit Jhala. Dependent types for javascript. *SIGPLAN Not.*, 47(10):587–606, October 2012. URL: `http:`

//doi.acm.org/10.1145/2398857.2384659, doi:10.1145/2398857.2384659.

[DDDG01] Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Fickle : Dynamic object re-classification. In Jorgen Lindskov Knudsen, editor, *ECOOP 2001 - Object-Oriented Programming, 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings*, volume 2072 of *Lecture Notes in Computer Science*, pages 130–149. Springer, 2001. URL: https://doi.org/10.1007/3-540-45337-7_8, doi:10.1007/3-540-45337-7_8.

[DDDG02] Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. More dynamic object reclassification: Fickle$_{||}$. *ACM Trans. Program. Lang. Syst.*, 24(2):153–191, 2002. URL: http://doi.acm.org/10.1145/514952.514955, doi:10.1145/514952.514955.

[DDG03] Ferruccio Damiani, Sophia Drossopoulou, and Paola Giannini. Refined effects for unanticipated object re-classification: Fickle$_3$. In Carlo Blundo and Cosimo Laneve, editors, *Theoretical Computer Science, 8th Italian Conference, ICTCS 2003, Bertinoro, Italy, October 13-15, 2003, Proceedings*, volume 2841 of *Lecture Notes in Computer Science*, pages 97–110. Springer, 2003. URL: https://doi.org/10.1007/978-3-540-45208-9_9, doi:10.1007/978-3-540-45208-9_9.

[DGHL98] Pietro Di Gianantonio, Furio Honsell, and Luigi Liquori. A lambda calculus of objects with self-inflicted extension. In Bjorn N. Freeman-Benson and Craig Chambers, editors, *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '98), Vancouver, British Columbia, Canada, October 18-22, 1998.*, pages 166–178. ACM, 1998. URL: http://doi.acm.org/10.1145/286936.286955, doi:10.1145/286936.286955.

[FHM94] 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.

[FM94] K. Fisher and J. C. Michell. Notes on Typed Object-Oriented Programming. In *Proc. of TACS*, volume 789, pages 844–885, 1994.

[FM95] K. Fisher and J. C. Mitchell. A Delegation-based Object Calculus with Subtyping. In *Proc. of FCT*, volume 965, pages 42–61, 1995.

[FM98] K. Fisher and J. C. Mitchell. On the relationship between classes, objects, and data abstraction. *Theory and Practice of Object Systems*, 1998. To appear.

[KT10] T Kamina and T Tamai. A smooth combination of role-based language and context activation. In *Proceedings of the Ninth Workshop on Foundation of Aspect-Oriented Languages (FOAL 2010),*, pages 15–24, 2010.

[Liq97] L. Liquori. An Extended Theory of Primitive Objects: First Order System. In *Proc. of ECOOP*, volume 1241, pages 146–169, 1997.

[Rém98] D. Rémy. From classes to objects via subtyping. In *Proc. of European Symposium on Programming*, volume 1381 of *lncs*. springer, 1998.

[RGN+11] Jorge Ressia, Tudor Gîrba, Oscar Nierstrasz, Fabrizio Perin, and Lukas Renggli. Talents: dynamically composable units of reuse. In Loïc Lagadec and Alain Plantec, editors, *Proceedings of the International Workshop on Smalltalk Technologies, IWST 2011, Edinburgh, United Kingdom, August 23, 2011*, pages 11:1–11:9. ACM, 2011. URL: `http://doi.acm.org/10.1145/2166929.2166940`, `doi:10.1145/2166929.2166940`.

[RGN+14] Jorge Ressia, Tudor Gîrba, Oscar Nierstrasz, Fabrizio Perin, and Lukas Renggli. Talents: an environment for dynamically composing units of reuse. *Softw., Pract. Exper.*, 44(4):413–432, 2014. URL: `http://dx.doi.org/10.1002/spe.2160`, `doi:10.1002/spe.2160`.

[RS98] J.G. Riecke and C. Stone. Privacy via Subsumption. In *Electronic proceedings of FOOL-98*, 1998.

[Ser99] Manuel Serrano. Wide classes. In Rachid Guerraoui, editor, *ECOOP'99 - Object-Oriented Programming, 13th European Conference, Lisbon, Portugal, June 14-18, 1999, Proceedings*, volume 1628 of *Lecture Notes in Computer Science*, pages 391–415. Springer, 1999. URL: `https://doi.org/10.1007/3-540-48743-3_18`, `doi:10.1007/3-540-48743-3_18`.

[Tak95] Masako Takahashi. Parallel reductions in lambda calculus. *Inf. Comput.*, 118(1):120–127, April 1995. URL: `http://dx.doi.org/10.1006/inco.1995.1057`, `doi:10.1006/inco.1995.1057`.

[Vou01] Jérôme Vouillon. Combining subsumption and binary methods: An object calculus with views. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 290–303, New York, NY, USA, 2001. ACM. URL: `http://doi.acm.org/10.1145/360204.360233`, `doi:10.1145/360204.360233`.

[Wan87] M. Wand. Complete Type Inference for Simple Objects. In *Proc. of LICS*, pages 37–44. IEEE Press, 1987.

[Zha10] Tian Zhao. Type inference for scripting languages with implicit extension. In *ACM SIGPLAN International Workshop on Foundations of Object-Oriented Languages*, 2010.

[Zha12] Tian Zhao. Polymorphic type inference for scripting languages with object extensions. *SIGPLAN Not.*, 47(2):37–50, October 2012. URL: `http://doi.acm.org/10.1145/2168696.2047855`, `doi:10.1145/2168696.2047855`.

About the authors