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

A prototype-based approach to object reclassification

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

a. DIMI, University of Udine, Italy

b. Université Côte d'Azur, Inria, 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 capability 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 the object itself (`this` in C^{++} and 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 the type of the inheriting objects. This feature

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

is usually referred to as "mytype method specialization". The 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 $\lambda\mathcal{O}bj$, dealing with 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].

Consider the method set_x belonging, among others, to a pt object with an x field:

$$pt \triangleq \langle x{=}\lambda s.0,\ \ set_x{=}\lambda s.\lambda v.\langle s \leftarrow x{=}\lambda s'.v\rangle,\ \ldots\rangle$$

When set_x is called to pt with argument "3", written as $pt \Leftarrow set_x(3)$, 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 the structure of a class can be modified only statically. If we need to add a new method to an instance of a class we are forced to recompile 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.

- *Downcasting* could be smoothly implementable on objects with self-extension methods. For example, for a colored point cpt extending the pt object above, the following expression could be made to type-check (details in Section 5):

$$cpt \Leftarrow eq(pt \Leftarrow add_{col}\,(black))$$

where add_{col} is intended to be a self-extension method of pt and eq is the name of the standard binary equality method.

- Self-extension is 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 $\lambda\mathcal{O}bj^{\oplus}$, 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 $\lambda\mathcal{O}bj^{\oplus}$ 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 $\leftarrow\!\!\oplus$, 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 1.1 Consider the add_{col} method, that adds a col field to the "point" object p:

$$p \triangleq \langle x{=}\lambda s.0,\;\; set_x{=}\lambda s.\lambda v.\langle s \leftarrow\!\!\oplus\, x{=}\lambda s'.v\rangle,\;\; add_{col}{=}\lambda s.\lambda v.\langle s \leftarrow\!\!\oplus\, col{=}\lambda s'.v\rangle\rangle$$

When add_{col} is sent to p with argument "white", i.e. $p \Leftarrow add_{col}(white)$, the result is a new object cp where the col field has been added to p and set to white:

$$cp \triangleq \langle x{=}\ldots,\;\; set_x{=}\ldots,\;\; add_{col}{=}\ldots,\;\; col{=}\lambda s.white\rangle$$

If add_{col} is sent twice to p, i.e. $cp \Leftarrow add_{col}(black)$, then, since the col field is already present in cp, it will be overwritten with the new "black" value.

As far as types are concerned, we add two new kinds of object-types, namely $\tau \oplus m$, which can be seen as the type theoretical counterpart of the syntactic object $\langle e_1 \leftarrow\!\oplus\, m = e_2 \rangle$, and $pro\,t.R \oplus m_1 \ldots \oplus m_k$, a generalization of the original $class\,t.R$ in [FHM94], named *pro*-type. Intuitively, if the type $pro\,t.R \oplus m_1 \ldots \oplus m_k$ is assigned to an object e (t represents the type of self), e can respond to all the methods $m_1, \ldots, m_k$. Mandatory, the list of pairs R contains all the methods $m_1, \ldots, m_k$ together with their 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 (therefore, if R did not contain reserved methods, $pro\,t.R \oplus m_1 \ldots \oplus m_k$ would coincide with $class\,t.R$ of [FHM94]).

To convey to the reader the intended meaning of *pro*-types, we suppose that an object e is assigned the type $pro\,t.\langle m{:}t \oplus n, n{:}int \rangle \oplus m$. In fact, $e \Leftarrow n$ is not typable, but as $e \Leftarrow m$ has the effect of adding the method n to the interface of e and of updating the type of e to $pro\,t.\langle m{:}t \oplus n, n{:}int \rangle \oplus m \oplus n$, then $(e \Leftarrow m) \Leftarrow n$ is typable.

The list of reserved methods in a *pro*-type is crucial to enforce the consistency of the type assignment system. Consider e.g. an object containing two methods, $addn_1$, and $addn_2$, 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, i.e. $obj\,t.R \oplus m_1 \ldots \oplus m_k$, which behaves like $pro\,t.R \oplus m_1 \ldots \oplus m_k$ except that it can be assigned to objects which can be extended only by making longer the list $\oplus m_1 \ldots \oplus m_k$ (by means of 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 $\lambda \mathcal{O}bj^{\oplus}$, 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 $\lambda \mathcal{O}bj^{\oplus}$ and 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. Section 5 presents a collection of typing examples. 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 with object reclassification. In Section 8 we discuss the related work. The complete set of type assignment rules appears in the Appendix, together with the full proofs.

[1] The *pro* and *obj* terminology is the same as in Fisher and Mitchell [FM95, FM98].

$$
\begin{array}{llcl}
(Beta) & (\lambda x.e_1)e_2 & \to & e_1[e_2/x] \\
(Selection) & e \Leftarrow m & \to & Sel(e, m, \lambda s.s) \\
(Success) & Sel(\langle e_1 \leftarrow\!\!\!\oplus\, m = e_2\rangle, m, e_3) & \to & e_2(e_3\langle e_1 \leftarrow\!\!\!\oplus\, m = e_2\rangle) \\
(Next) & Sel(\langle e_1 \leftarrow\!\!\!\oplus\, n = e_2\rangle, m, e_3) & \to & Sel(e_1, m, \lambda s.e_3\langle s \leftarrow\!\!\!\oplus\, n = e_2\rangle)
\end{array}
$$

Figure 1 – Reduction Semantics (Small-Step)

The present work extends and completes [DGHL98] in the following way: 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 $\lambda\mathcal{O}bj^{\oplus}$. The terms are defined by the following abstract grammar:

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

where c, x, m are meta-variables ranging over sets of constants, variables, and names of methods, respectively. As usual, terms that differ only in the names of bound variables are identified. Terms are untyped λ-terms enriched with objects: the intended meaning of the object-terms is the following: $\langle\,\rangle$ stands for the empty object; $\langle e_1 \leftarrow\!\!\!\oplus\, m = e_2\rangle$ stands for extending/overriding the object e_1 with a method m whose body is e_2; $e \Leftarrow m$ stands for the result of sending the message m to the object e.

The auxiliary operation $Sel(e_1, m, e_2)$ searches the body of the m method within the object e_1. In the recursive search of m, $Sel(e_1, m, e_2)$ 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.

To lighten up the notation, we write $\langle m_1{=}e_1, \ldots, m_k{=}e_k\rangle$ as syntactic sugar for $\langle \ldots \langle\langle\rangle \leftarrow\!\!\!\oplus\, m_1{=}e_1\rangle \ldots \leftarrow\!\!\!\oplus\, m_k{=}e_k\rangle$, where $k{\geq}1$. Also, we write e in place of $\lambda x.e$ if $x{\notin}FV(e)$; this mainly concerns methods, whose first formal parameter is always their host object: e.g. $\lambda s.1$ and $\lambda s'.(s \Leftarrow m)$ are usually written 1 and $s \Leftarrow m$, respectively.

2.1 Operational semantics

We define the semantics of $\lambda\mathcal{O}bj^{\oplus}$ terms by means of the reduction rules displayed in Figure 1 (small-step semantics $\to$); the evaluation relation $\twoheadrightarrow$ is then taken to be the symmetric, reflexive, transitive and contextual closure of $\to$.

In addition to the standard β-rule for λ-calculus, the main operation on objects is method invocation, whose reduction is defined by the ($Select$) rule. Sending a message m to an object e which contains a method m reduces to $Sel(e, m, \lambda s.s)$, 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 original receiver.

By looking at the last two rules, one may note that the Sel function scans the receiver of the message until it finds the definition of the called method: when it finds such a method, it applies its body to the receiver 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; the following reduction illustrates the evaluation mechanism:

$$\begin{array}{lll}
\langle id = \lambda s.s,\ one = 1\rangle \Leftarrow id & \to & \\
Sel(\langle id = \lambda s.s,\ one = 1\rangle,\ id,\ \lambda s'.s') & \to & \\
Sel(\langle id = \lambda s.s\rangle,\ id,\ \lambda s''.(\lambda s'.s')\langle s'' \leftarrow\!\!\!\oplus\, one = 1\rangle) & \to & \\
Sel(\langle id = \lambda s.s\rangle,\ id,\ \lambda s''.\langle s'' \leftarrow\!\!\!\oplus\, one = 1\rangle) & \to & \\
(\lambda s.s)((\lambda s''.\langle s'' \leftarrow\!\!\!\oplus\, one = 1\rangle)\langle id = \lambda s.s\rangle) & \to & \langle id = \lambda s.s,\ one = 1\rangle
\end{array}$$

That is, in order to call the first method id of an object-term with two methods, $\langle id = \lambda s.s,\ one = 1\rangle$, one needs to consider the subterm containing just the first method $\langle id = \lambda s.s\rangle$ and construct a function, $\lambda s''.\langle s'' \leftarrow\!\!\!\oplus\, one = 1\rangle$, transforming the subterm in the original term.

Proposition 2.1 *The $\to$ 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 on λ-terms, where several redexes can be reduced in parallel; then one shows that for any term e there is a term e^*, i.e. Takahashi's translation, obtained from M by reducing a maximum set of redexes in parallel. It follows almost immediately that the parallel reduction satisfies the triangular property, hence the diamond property, and therefore the calculus in confluent. With respect to the λ-calculus, $\lambda\mathcal{O}bj^{\oplus}$ contains, besides the λ-rule, reduction rules for object terms; however, the latter do not interfere with the former, hence Takahashi's technique can be straightforwardly applied to the $\lambda\mathcal{O}bj^{\oplus}$ calculus.

A deterministic, call by name, evaluation strategy over terms $\stackrel{det}{\to}$ may be defined on $\lambda\mathcal{O}bj^{\oplus}$ 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:

$$C[\ ] = [\ ] \mid C[\ ]e \mid C[\ ] \Leftarrow m \mid Sel(C[\ ], m, e)$$

The set of values, i.e. the terms that are well-formed (typable according to the next defined type system) and where no reduction is possible, is defined by the following grammar:

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

2.2 Examples

In the next examples we show three objects, performing, respectively:

- a 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:

$$self_{ext} \quad \triangleq \quad \langle add_n = \lambda s.\langle s \leftarrow\!\oplus\, n{=}1\rangle\rangle.$$

If we send the message add_n to $self_{ext}$, then we have the following computation:

$$\begin{array}{rcl} self_{ext} \Leftarrow add_n & \to & Sel(self_{ext}, add_n, \lambda s'.s') \\ & \twoheadrightarrow & (\lambda s.\langle s \leftarrow\!\oplus\, n{=}1\rangle)self_{ext} \\ & \to & \langle self_{ext} \leftarrow\!\oplus\, n{=}1\rangle \end{array}$$

i.e. the method n has been added to $self_{ext}$. If we send the message add_n twice to $self_{ext}$, i.e. $\langle self_{ext} \leftarrow\!\oplus\, n{=}1\rangle \Leftarrow add_n$, 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:

$$inner_{ext} \triangleq \langle add_{mn} = \lambda s.\langle s \leftarrow\!\oplus\, m{=}\lambda s'.\langle s' \leftarrow\!\oplus\, n{=}1\rangle\rangle\rangle$$

If we send the message add_{mn} to $inner_{ext}$, then we obtain:

$$inner_{ext} \Leftarrow add_{mn} \twoheadrightarrow \langle inner_{ext} \leftarrow\!\oplus\, m{=}\lambda s.\langle s \leftarrow\!\oplus\, n{=}1\rangle\rangle$$

i.e. the method m has been added to $inner_{ext}$. On the other hand, if we send first the message add_{mn} and then m to $inner_{ext}$, both the methods m and n are added:

$$(inner_{ext} \Leftarrow add_{mn}) \Leftarrow m \quad \twoheadrightarrow \quad \begin{array}{lcl} \langle add_{mn} & = & \lambda s.\langle s \leftarrow\!\oplus\, m{=}\lambda s'.\langle s' \leftarrow\!\oplus\, n{=}1\rangle\rangle, \\ \;\; m & = & \lambda s.\langle s \leftarrow\!\oplus\, n{=}1\rangle, \\ \;\; n & = & 1\rangle \end{array}$$

Example 2.3 Consider the object fly_{ext}, defined as follows:

$$fly_{ext} \triangleq \langle f{=}\lambda s.\lambda s'.s' \Leftarrow n,\ get_f{=}\lambda s.(s \Leftarrow f)\langle s \leftarrow\!\oplus\, n{=}1\rangle\rangle$$

If we send the message get_f to fly_{ext}, then we get the following computation:

$$\begin{array}{rcl} fly_{ext} \Leftarrow get_f & \to & Sel(fly_{ext}, get_f, \lambda s''.s'') \\ & \to & (\lambda s.(s \Leftarrow f)\langle s \leftarrow\!\oplus\, n{=}1\rangle)fly_{ext} \\ & \to & (fly_{ext} \Leftarrow f)\langle fly_{ext} \leftarrow\!\oplus\, n{=}1\rangle \\ & \to & Sel(fly_{ext}, f, \lambda s''.s'')\langle fly_{ext} \leftarrow\!\oplus\, n{=}1\rangle \\ & \twoheadrightarrow & (\lambda s.\lambda s'.s' \Leftarrow n)fly_{ext}\langle fly_{ext} \leftarrow\!\oplus\, n{=}1\rangle \\ & \twoheadrightarrow & \langle fly_{ext} \leftarrow\!\oplus\, n{=}1\rangle \Leftarrow n \\ & \twoheadrightarrow & 1 \end{array}$$

i.e. the following steps are performed:

1. the method get_f calls the method f with actual parameter the host 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 constant 1.

3 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:

$$
\begin{array}{rcll}
\sigma & ::= & \iota \mid \sigma{\to}\sigma \mid \tau & \text{(generic-types)} \\
\tau & ::= & t \mid pro\,t.R \mid \tau \oplus m & \text{(object-types)} \\
R & ::= & \langle\rangle \mid \langle R, m{:}\sigma\rangle & \text{(rows)} \\
\kappa & ::= & * & \text{(kind of types)}
\end{array}
$$

In the rest of the article we will use σ as meta-variable ranging over generic-types, ι over constant types, τ over object-types. Moreover, t is a type variable, R a metavariable ranging over rows, i.e. unordered sets of pairs *(method label, method type)*, m a method label, and κ a metavariable ranging over the unique kind of types $*$.

To ease the notation, we write $\langle \ldots \langle\langle\rangle, m_1{:}\sigma_1\rangle \ldots, m_k{:}\sigma_k\rangle$ as $\langle m_1{:}\sigma_1, \ldots, m_k{:}\sigma_k\rangle$ or $\langle \overline{m}_k{:}\overline{\sigma}_k\rangle$ or else simply $\langle \overline{m}{:}\overline{\sigma}\rangle$ in the case the subscripts can be omitted. Similarly, we write either $\tau \oplus \overline{m}_k$ or $\tau \oplus \overline{m}$ for $\tau \oplus m_1 \ldots \oplus m_k$. If $R \equiv \langle \overline{m}{:}\overline{\sigma}\rangle$, then we denote $\overline{m}$ by $\overline{R}$, and we write $R_1 \subseteq R_2$ if $R_1 \equiv \langle \overline{m}{:}\overline{\sigma}_1\rangle$ and $R_2 \equiv \langle \overline{m}{:}\overline{\sigma}_1, \overline{n}{:}\overline{\sigma}_2\rangle$.

As in [FHM94], we may consider object-types as a form of recursively-defined types. Object-types in the form $pro\,t.R \oplus \overline{m}$ are named *pro*-types, where *pro* is a binder for the type-variable t representing “self” (we use α-conversion of type-variables bound by *pro*). The intended meaning of a *pro*-type $pro\,t.\langle \overline{m}{:}\overline{\sigma}\rangle \oplus \overline{n}$ is the following:

- the methods in $\overline{m}$ are the ones which are present in the *pro*-type;
- the methods in $\overline{n}$, being in fact a subset of those in $\overline{m}$, are the methods that are *available* and can be invoked (it follows that the *pro*-type $pro\,t.\langle \overline{m}{:}\overline{\sigma}\rangle \oplus \overline{m}$ corresponds exactly to the object-type $class\,t.\langle \overline{m}{:}\overline{\sigma}\rangle$ in [FHM94]);
- the methods in $\overline{m}$ that do not appear in $\overline{n}$ are methods that cannot be invoked: they are just *reserved*.

In the end, we can say that the operator “$\oplus$” is used to make active and usable those methods that were previously just reserved in a *pro*-type; essentially, $\oplus$ is the “type counterpart” of the operator on terms $\leftarrow\!\!\!\oplus$. In the following, it will turn out that we can extend an object e with a new method m having type σ only if it is possible to assign to e an object-type of the form $pro\,t.\langle R, m{:}\sigma\rangle \oplus \overline{n}, m$; this reservation mechanism is crucial to guarantee the consistency of the type system.

3.2 Contexts and judgments

The contexts have the following form:

$$\Gamma ::= \varepsilon \mid \Gamma, x{:}\sigma \mid \Gamma, t \prec\!\!\!\!\!\!\;\sharp\; \tau$$

Our type assignment system uses judgments of the following shapes:

$$\Gamma \vdash ok \qquad \Gamma \vdash \sigma : * \qquad \Gamma \vdash e : \sigma \qquad \Gamma \vdash \tau_1 \mathrel{-\!\!\!\triangleleft\!\!\!\sharp}\, \tau_2$$

The intended meaning of the first three judgments is standard: well-formed contexts and types, and assignment of type σ to term e. The intended meaning of $\Gamma \vdash \tau_1 \mathrel{-\!\!\!\triangleleft\!\!\!\sharp}\, \tau_2$ is that τ_1 is the type of a possible extension of an object having type τ_2. As in [Bru94], and in [BL95, BBL96, BBDL97, 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.

3.3 Well formed context and types

The type rules for well-formed contexts are quite standard. We just comment that in the $(Cont{-}t)$ rule

$$\frac{\Gamma \vdash prot.R \oplus \overline{m} : * \quad t \notin Dom(\Gamma)}{\Gamma, t \mathrel{-\!\!\!\triangleleft\!\!\!\sharp}\, prot.R \oplus \overline{m} \vdash ok}$$

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.

The $(Type{-}Pro)$ rule

$$\frac{\Gamma, t \mathrel{-\!\!\!\triangleleft\!\!\!\sharp}\, prot.R \vdash \sigma : * \quad m \notin \overline{R}}{\Gamma \vdash prot.\langle R, m{:}\sigma\rangle : *}$$

asserts that the object-type $prot.\langle R, m{:}\sigma\rangle$ is well-formed if the object-type $prot.R$ is well-formed and the type σ is well-formed under the hypothesis that t is an object-type containing the methods in $\overline{R}$. Since σ may contain a subexpression in the form $t \oplus n$, with $n \in \overline{R}$, we need to introduce in the context the hypothesis $t \mathrel{-\!\!\!\triangleleft\!\!\!\sharp}\, prot.R$ to prove that $t \oplus n$ is a well-formed type.

The $(Type{-}Extend)$ rule

$$\frac{\Gamma \vdash \tau \mathrel{-\!\!\!\triangleleft\!\!\!\sharp}\, prot.R \quad \overline{m} \subseteq \overline{R}}{\Gamma \vdash \tau \oplus \overline{m} : *}$$

asserts that in order to activate the methods $\overline{m}$ in the object-type τ, the methods $\overline{m}$ need to be present (reserved) in τ.

3.4 Matching rules

The $(Match{-}Pro)$ rule

$$\frac{\Gamma \vdash prot.R_1 \oplus \overline{m} : * \quad \Gamma \vdash prot.R_2 \oplus \overline{n} : * \quad R_2 \subseteq R_1 \quad \overline{n} \subseteq \overline{m}}{\Gamma \vdash prot.R_1 \oplus \overline{m} \mathrel{-\!\!\!\triangleleft\!\!\!\sharp}\, prot.R_2 \oplus \overline{n}}$$

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

$$\frac{\Gamma_1, t \mathrel{-\!\!\!\triangleleft\!\!\!\sharp}\, \tau_1, \Gamma_2 \vdash \tau_1 \oplus \overline{m} \mathrel{-\!\!\!\triangleleft\!\!\!\sharp}\, \tau_2}{\Gamma_1, t \mathrel{-\!\!\!\triangleleft\!\!\!\sharp}\, \tau_1, \Gamma_2 \vdash t \oplus \overline{m} \mathrel{-\!\!\!\triangleleft\!\!\!\sharp}\, \tau_2}$$

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 τ_1, and τ_1 itself, incremented with a set of methods $\overline{m}$, specializes a type τ_2, then, by transitivity of the matching relation, t, incremented by the methods in $\overline{m}$, specializes τ_2.

The $(Match{-}t)$ rule

$$\frac{\Gamma \vdash t \oplus \overline{m} : * \quad \overline{n} \subseteq \overline{m}}{\Gamma \vdash t \oplus \overline{m} \mathrel{-\!\!\!\lhd\!\!\sharp} t \oplus \overline{n}}$$

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

3.5 Terms rules

The type rules for λ-terms are self-explanatory and hence they need no further justification. Concerning those for object terms, the $(Empty)$ rule assigns to an empty object an empty *pro*-type, while the $(Pre{-}Extend)$ rule

$$\frac{\Gamma \vdash e : prot.R_1 \oplus \overline{m} \quad \Gamma \vdash prot.\langle R_1, R_2\rangle \oplus \overline{m} : *}{\Gamma \vdash e : prot.\langle R_1, R_2\rangle \oplus \overline{m}}$$

asserts that an object e having type $prot.R_1 \oplus \overline{m}$ can be considered also an object having type $prot.\langle R_1, R_2\rangle \oplus \overline{m}$, 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 a variable s representing self; in fact, as explained in the Remark 3.1 below, the type of s can only be a type variable. This fact is crucial for the soundness of the type system.

The $(Extend)$ rule

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash \tau \mathrel{-\!\!\!\lhd\!\!\sharp} prot.\langle R, n{:}\sigma\rangle \oplus \overline{m} \\ \Gamma, t \mathrel{-\!\!\!\lhd\!\!\sharp} prot.\langle R, n{:}\sigma\rangle \oplus \overline{m}, n \vdash e_2 : t {\to} \sigma\end{array}}{\Gamma \vdash \langle e_1 \leftarrow\!\!\!\oplus\, n = e_2\rangle : \tau \oplus n}$$

can be applied in the following cases:

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

2. when τ is a type variable t. In this case e_1 can be the variable s, and a *self-inflicted extension* takes place.

The bound for t is the same as the final type for the object $\langle e_1 \leftarrow\!\!\!\oplus\, n = e_2\rangle$; this allows a recursive call of the method n inside the expression e_2, defining the method n itself.

The $(Override)$ rule

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash \tau \mathrel{-\!\!\!\lhd\!\!\sharp} prot.\langle R, n{:}\sigma\rangle \oplus \overline{m}, n \\ \Gamma, t \mathrel{-\!\!\!\lhd\!\!\sharp} prot.\langle R, n{:}\sigma\rangle \oplus \overline{m}, n \vdash e_2 : t {\to} \sigma\end{array}}{\Gamma \vdash \langle e_1 \leftarrow\!\!\!\oplus\, n = e_2\rangle : \tau}$$

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

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

The ($Send$) rule

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau \mathrel{-\!\!\lhd\!\!\!\!\sharp}\, prot.\langle R, n{:}\sigma\rangle \oplus \overline{m}, n}{\Gamma \vdash e \Leftarrow n : \sigma[\tau/t]}$$

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

In the ($Select$) rule

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash \tau \mathrel{-\!\!\lhd\!\!\!\!\sharp}\, prot.\langle R, n{:}\sigma\rangle \oplus \overline{m}, n \\ \Gamma, t \mathrel{-\!\!\lhd\!\!\!\!\sharp}\, prot.\langle R, n{:}\sigma\rangle \oplus \overline{m}, n \vdash e_2 : t \to t \oplus \overline{n}\end{array}}{\Gamma \vdash Sel(e_1, n, e_2) : \sigma[\tau \oplus \overline{n}/t]}$$

the first two conditions ensure that the n method is available in e_1, while the last one that e_2 is a function that transforms an object into a more refined one.

4 Dealing with object subsumption

While the type assignment system $\lambda\mathcal{O}bj^{\oplus}$, presented in Section 3, allows self-inflicted extension, it does not allow object subsumption. This is not surprising: 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. The papers [AC96, FM94, FHM94, BL95] propose different type systems for prototype-based languages, where subsumption is permitted only in absence of object extension (and a fortiori self-inflicted extension). In this section, we devise a conservative extension of $\lambda\mathcal{O}bj^{\oplus}$, that we name $\lambda\mathcal{O}bj^{\oplus}_S$, to accommodate width-subtyping.

In the perspective of adding a subsumption rule to the typing system, we introduce another kind of object-types, i.e. $obj\,t.R \oplus \overline{m}$, named *obj*-types. The main difference between the *pro*-types and the *obj*-types consists in the fact that the ($Pre{-}Extend$) rule cannot be applied when an object has type $obj\,t.R \oplus \overline{m}$; it follows that the type $obj\,t.R \oplus \overline{m}$ permits extensions of an object only by enriching the list $\overline{m}$, i.e. by making active its reserved methods. This approach to subsumption is inspired by the one in [FM95, Liq97]. Formally, we need to extend the syntax of types by means of *obj*-types and the kind of *rigid*, i.e. non-extensible, types:

$$\tau \quad ::= \quad \ldots \mid obj\,t.R \qquad \text{(object-types)}$$

$$\kappa \quad ::= \quad \ldots \mid *_{rgd} \qquad \text{(kind of types)}$$

The subset of rigid types contains the *obj*-types and is closed under the arrow constructor. In order to axiomatize this, we introduce the judgment $\Gamma \vdash \tau : *_{rgd}$, whose rules are reported in Appendix B. Intuitively, we can use the matching relation as a subtyping relation only when the type in the conclusion is rigid:

$$\frac{\Gamma \vdash e : \tau_1 \quad \Gamma \vdash \tau_1 \mathrel{-\!\!\lhd\!\!\!\!\sharp}\, \tau_2 \qquad \Gamma \vdash \tau_2 : *_{rgd}}{\Gamma \vdash e : \tau_2} \; (Subsume)$$

Let be $\tau \triangleq pro\,t.\langle add_n{:}t \oplus n,\ n{:}int\rangle$ and $\Gamma \triangleq t \prec\!\!\!\sharp\, \tau \oplus add_n, s{:}t$. Then:

$$\dfrac{\dfrac{\vdots}{\vdash \langle\,\rangle : \tau} \quad \dfrac{\vdots}{\vdash \tau \prec\!\!\!\sharp\, \tau} \quad \Delta}{\vdash \langle add_n = \lambda s.\langle s \leftarrow\!\!\oplus\, n = 1\rangle\rangle : \tau \oplus add_n}\ (Extend)$$

where the first two premises are derived straightforwardly and Δ as follows:

$$\dfrac{\dfrac{\Gamma \vdash s : t \quad \Gamma \vdash t \prec\!\!\!\sharp\, pro\,t.\langle n{:}int\rangle \quad \Gamma, t' \prec\!\!\!\sharp\, pro\,t'.\langle n{:}int\rangle \oplus n \vdash 1 : t' \to int}{\Gamma \vdash \langle s \leftarrow\!\!\oplus\, n = 1\rangle : t \oplus n}\ (Extend)}{t \prec\!\!\!\sharp\, \tau \oplus add_n \vdash \lambda s.\langle s \leftarrow\!\!\oplus\, n = 1\rangle : (t \to t \oplus n)}\ (Abs)$$

Figure 2 – A derivation for $self_{ext}$

This is in fact is the rule performing object subsumption: it allows to use objects with an extended signature in any context expecting objects with a shorter one.

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, in our type assignment system, the matching is a relation on types compatible with a limited subsumption rule.

Most of the rules for *obj*-types are a rephrasing of the rules presented so far, replacing the binder *pro* with *obj*. We remark that the $(Type{-}Obj{-}Rdg)$ rule

$$\dfrac{\Gamma \vdash obj\,t.\langle \overline{m}_k : \overline{\sigma}_k\rangle \oplus \overline{n} : * \quad \forall i \le k.\ \Gamma \vdash \sigma_i : *_{rgd} \wedge t \text{ covariant in } \sigma_i}{\Gamma \vdash obj\,t.\langle \overline{m}_k : \overline{\sigma}_k\rangle \oplus \overline{n} : *_{rgd}}$$

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 $\overline{\sigma}_k$. A natural (and sound) consequence is that we cannot forget binary methods via subtyping (see [BCC+96, Cas95, Cas96]). The $(Promote)$ rule

$$\dfrac{\Gamma \vdash pro\,t.R_1 \oplus \overline{m} : * \quad \Gamma \vdash obj\,t.R_2 \oplus \overline{n} : * \quad R_2 \subseteq R_1 \quad \overline{n} \subseteq \overline{m}}{\Gamma \vdash pro\,t.R_1 \oplus \overline{m} \prec\!\!\!\sharp\, obj\,t.R_2 \oplus \overline{n}}$$

promotes a fully-specializable *pro*-type into a limitedly specializable *obj*-type with less reserved and less available methods.

5 Examples

In this section, we give the types of the examples presented in Section 2.2, together with some other 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:

$$\begin{array}{rcl} self_{ext} & : & pro\,t.\langle add_n{:}t \oplus n,\ n{:}int\rangle \oplus add_n \\ inner_{ext} & : & pro\,t.\langle add_{mn}{:}t \oplus m,\ m{:}t \oplus n,\ n{:}int\rangle \oplus add_{mn} \\ fly_{ext} & : & pro\,t.\langle f{:}t \oplus n,\ get_f{:}t \oplus n \to int,\ n{:}int\rangle \oplus f, get_f \end{array}$$

A possible derivation for $self_{ext}$ is presented in Figure 2.

Example 5.1 We show how class declaration can be simulated in $\lambda\mathcal{O}bj^{\oplus}$ 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_{col} be defined as in Example 1.1, and let us consider the simple class definition:

$$P_{class} \triangleq \langle new = \lambda s.\langle n{=}1,\ add_{col}{=}\lambda s'.\lambda x.\langle s' \leftarrow\!\!\!\oplus\, col{=}x\rangle\rangle\rangle$$

Then, the object P_{class} can be used to create instances of both points and colored points, by using the expressions:

$$P_{class} \Leftarrow new \qquad and \qquad (P_{class} \Leftarrow new) \Leftarrow add_{col}(white)$$

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

$$\begin{array}{rcl} P & \triangleq & obj\,t.\langle n{:}int,\ col{:}colors\rangle \oplus n \\ CP & \triangleq & obj\,t.\langle n{:}int,\ col{:}colors\rangle \oplus n, col \\ g & \triangleq & \lambda s.\langle s \leftarrow\!\!\!\oplus\, col = white\rangle \end{array}$$

and let p and cp be of type P and CP, respectively. Then, we can derive:

$$\begin{array}{l} \vdash CP \mathrel{\triangleleft\!\!\!\sharp} P \qquad \vdash g : P{\to}CP \qquad \vdash g(cp) : CP \\ \vdash (\lambda f.equal(f(p) \Leftarrow col, f(cp) \Leftarrow col))g : bool \end{array}$$

where the equality function equal has type $t{\to}t{\to}bool$. Notice that the terms:

$$g(cp) \qquad (\lambda f.equal(f(p) \Leftarrow col, f(cp) \Leftarrow col))$$

would not be typable without the subsumption rule.

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

$$\begin{array}{rcl} Q & \triangleq & obj\,t.\langle n{:}int\rangle \oplus n \\ q & \triangleq & \langle copy_n = \lambda s.\lambda s'.\langle s \leftarrow\!\!\!\oplus\, n = s' \Leftarrow n\rangle\rangle \end{array}$$

By assuming p and cp as in Example 5.2, we can derive:

$$\begin{array}{rcl} \vdash q & : & pro\,t.\langle copy_n{:}Q \to t \oplus n,\ n{:}int\rangle \oplus copy_n \\ \vdash q \Leftarrow copy_n(cp) & : & pro\,t.\langle n{:}int,\ copy_n{:}Q \to t\rangle \oplus n, copy_n \\ \vdash q \Leftarrow copy_n(cp) \Leftarrow copy_n(p) & : & pro\,t.\langle n{:}int,\ copy_n{:}Q \to t\rangle \oplus n, copy_n \end{array}$$

Notice in particular that the object $q \Leftarrow copy_n(cp) \Leftarrow copy_n(p)$ 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 p_1 and cp_1 be objects with eq methods (checking the values of n and the pairs (n, col), respectively), and add_{col} the self-extension method presented in Example 5.1, typable as follows:

$$\vdash p_1 : pro\,t.R \qquad and \qquad \vdash cp_1 : pro\,t.R \oplus col$$

where $R \triangleq \langle n{:}int,\ eq{:}t{\to}bool,\ add_{col}{:}colors \to t{\oplus}col,\ col{:}colors\rangle{\oplus}n, eq, add_{col}$. *Then, the following judgments are derivable:*

$$\begin{array}{rcl} \vdash cp_1 \Leftarrow eq & : & prot.R \oplus col \to bool \\ \vdash p_1 \Leftarrow add_{col}(white) & : & prot.R \oplus col \\ \vdash cp_1 \Leftarrow eq(p_1 \Leftarrow add_{col}(white)) & : & bool \end{array}$$

6 Soundness of the Type System

In this section, we prove the crucial property of our type system, i.e. the Subject Reduction theorem 6.9. 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 Theorem 6.9, we shall derive the fundamental result of the paper, i.e. the Type Soundness of our typing discipline. The proofs are fully documented in Appendixes C and D.

We first address the plain type assignment system without subsumption $\lambda\mathcal{O}bj^{\oplus}$, then in Section 6.1 we extend the Subject Reduction to the whole type system $\lambda\mathcal{O}bj^{\oplus}_{S}$. In the presentation of the formal results, we need α, β as metavariables for generic-types and ρ, υ for object-types. Moreover, $\mathcal{A}$ is a metavariable ranging on statements in the forms ok, $\alpha : *$, $\upsilon \mathrel{-\!\!\!\sharp} \rho$, $e : \beta$, and $\mathcal{C}$ on statements in the forms $x{:}\sigma$, $t \mathrel{-\!\!\!\sharp} \tau$.

Lemma 6.1 *(Sub-derivation)*

(i) If Δ *is a derivation of* $\Gamma_1, \Gamma_2 \vdash \mathcal{A}$*, then there exists a sub-derivation* $\Delta' \subseteq \Delta$ *of* $\Gamma_1 \vdash ok$.

(ii) If Δ *is a derivation of* $\Gamma_1, x{:}\sigma, \Gamma_2 \vdash \mathcal{A}$*, then there exists a sub-derivation* $\Delta' \subseteq \Delta$ *of* $\Gamma_1 \vdash \sigma : *$.

(iii) If Δ *is a derivation of* $\Gamma_1, t \mathrel{-\!\!\!\sharp} \tau, \Gamma_2 \vdash \mathcal{A}$*, then there exists a sub-derivation* $\Delta' \subseteq \Delta$ *of* $\Gamma_1 \vdash \tau : *$.

Lemma 6.2 *(Weakening)*

(i) If $\Gamma_1, \Gamma_2 \vdash \mathcal{A}$ *and* $\Gamma_1, \mathcal{C}, \Gamma_2 \vdash ok$*, then* $\Gamma_1, \mathcal{C}, \Gamma_2 \vdash \mathcal{A}$.

(ii) If $\Gamma_1 \vdash \mathcal{A}$ *and* $\Gamma_1, \Gamma_2 \vdash ok$*, then* $\Gamma_1, \Gamma_2 \vdash \mathcal{A}$.

Lemma 6.3 *(Well-formed object-types)*

(i) $\Gamma \vdash prot.R \oplus \overline{m} : *$ *if and only if* $\Gamma \vdash prot.R : *$ *and* $\overline{m} \subseteq \overline{R}$.

(ii) $\Gamma \vdash t \oplus \overline{m} : *$ *if and only if* Γ *contains* $t \mathrel{-\!\!\!\sharp} prot.R \oplus \overline{n}$*, with* $\overline{m} \subseteq \overline{R}$.

Proposition 6.4 *(Matching is well-formed)*

If $\Gamma \vdash \tau_1 \mathrel{-\!\!\!\sharp} \tau_2$*, then* $\Gamma \vdash \tau_1 : *$ *and* $\Gamma \vdash \tau_2 : *$.

Lemma 6.5 *(Matching)*

(i) $\Gamma \vdash prot.R_1 \oplus \overline{m} \mathrel{-\!\!\!\sharp} \tau_2$ *if and only if* $\Gamma \vdash prot.R_1 \oplus \overline{m} : *$ *and* $\Gamma \vdash \tau_2 : *$ *and* $\tau_2 \equiv prot.R_2 \oplus \overline{n}$*, with* $R_2 \subseteq R_1$ *and* $\overline{n} \subseteq \overline{m}$.

(ii) $\Gamma \vdash \tau_1 \mathrel{-\!\!\!\sharp} t \oplus \overline{n}$ *if and only if* $\Gamma \vdash \tau_1 : *$ *and* $\tau_1 \equiv t \oplus \overline{m}$*, with* $\overline{n} \subseteq \overline{m}$.

(iii) $\Gamma \vdash t \oplus \overline{m} \mathrel{\prec\!\!\!\#} prot.R_2 \oplus \overline{n}$ *if and only if* Γ *contains* $t \mathrel{\prec\!\!\!\#} prot.R_1 \oplus \overline{p}$*, with* $R_2 \subseteq R_1$ *and* $\overline{n} \subseteq \overline{m} \cup \overline{p}$*.*

(iv) (Reflexivity) If $\Gamma \vdash \rho : *$ *then* $\Gamma \vdash \rho \mathrel{\prec\!\!\!\#} \rho$*.*

(v) (Transitivity) If $\Gamma \vdash \tau_1 \mathrel{\prec\!\!\!\#} \rho$ *and* $\Gamma \vdash \rho \mathrel{\prec\!\!\!\#} \tau_2$*, then* $\Gamma \vdash \tau_1 \mathrel{\prec\!\!\!\#} \tau_2$*.*

(vi) (Uniqueness) If $\Gamma \vdash \tau_1 \mathrel{\prec\!\!\!\#} prot.\langle R_1, m{:}\sigma_1\rangle$ *and* $\Gamma \vdash \tau_1 \mathrel{\prec\!\!\!\#} prot.\langle R_2, m{:}\sigma_2\rangle$*, then* $\sigma_1 \equiv \sigma_2$*.*

(vii) If $\Gamma \vdash \tau_1 \mathrel{\prec\!\!\!\#} \tau_2$ *and* $\Gamma \vdash \tau_2 \oplus m : *$*, then* $\Gamma \vdash \tau_1 \oplus m \mathrel{\prec\!\!\!\#} \tau_2 \oplus m$*.*

(viii) If $\Gamma \vdash \tau_1 \oplus m \mathrel{\prec\!\!\!\#} prot.R \oplus \overline{n}$*, then* $\Gamma \vdash \tau_1 \mathrel{\prec\!\!\!\#} prot.R \oplus \overline{n} - m$*.*

(ix) If $\Gamma \vdash \rho \oplus m : *$*, then* $\Gamma \vdash \rho \oplus m \mathrel{\prec\!\!\!\#} \rho$*.*

Lemma 6.6 *(Match Weakening)*

(i) If $\Gamma_1, t \mathrel{\prec\!\!\!\#} \rho, \Gamma_2 \vdash \mathcal{A}$ *and* $\Gamma_1 \vdash \tau \mathrel{\prec\!\!\!\#} \rho$*, with* τ *a* `pro`*-type, then* $\Gamma_1, t \mathrel{\prec\!\!\!\#} \tau, \Gamma_2 \vdash \mathcal{A}$*.*

(ii) If $\Gamma \vdash prot.\langle R, n{:}\sigma\rangle \oplus \overline{m} : *$*, then* $\Gamma, t \mathrel{\prec\!\!\!\#} prot.\langle R, n{:}\sigma\rangle \oplus \overline{m} \vdash \sigma : *$*.*

Proposition 6.7 *(Substitution)*

(i) If $\Gamma_1, x{:}\sigma, \Gamma_2 \vdash \mathcal{A}$ *and* $\Gamma_1 \vdash e : \sigma$*, then* $\Gamma_1, \Gamma_2 \vdash \mathcal{A}[e/x]$*.*

(ii) If $\Gamma_1, t \mathrel{\prec\!\!\!\#} \tau, \Gamma_2, \Gamma_3 \vdash \mathcal{A}$ *and* $\Gamma_1, t \mathrel{\prec\!\!\!\#} \tau, \Gamma_2 \vdash \rho \mathrel{\prec\!\!\!\#} \tau$*, then* $\Gamma_1, t \mathrel{\prec\!\!\!\#} \tau, \Gamma_2, \Gamma_3[\rho/t] \vdash \mathcal{A}[\rho/t]$*.*

(iii) If $\Gamma_1, t \mathrel{\prec\!\!\!\#} \tau, \Gamma_2 \vdash \mathcal{A}$ *and* $\Gamma_1 \vdash \rho \mathrel{\prec\!\!\!\#} \tau$*, then* $\Gamma_1, \Gamma_2[\rho/t] \vdash \mathcal{A}[\rho/t]$*.*

Proposition 6.8 *(Types of expressions are well-formed)*

If $\Gamma \vdash e : \beta$*, then* $\Gamma \vdash \beta : *$*.*

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

Theorem 6.9 *(Subject Reduction,* $\lambda\mathcal{O}bj^{\oplus}$*) If* $\Gamma \vdash e : \beta$ *and* $e \to e'$*, then* $\Gamma \vdash e' : \beta$*.*

We proceed by deriving the Type Soundness theorem: it guarantees, 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 method m into an expression that does not reduce to an object which has the method m in its interface.

Definition 6.10 *We define the set of wrong terms as follows:*

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

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* $\varepsilon \vdash e : \beta$*, then* $e \not\twoheadrightarrow C[wrong]$*, where* $C[\ ]$ *is a generic context in* $\lambda\mathcal{O}bj^{\oplus}$*, i.e. a term with an "hole" inside it.*

6.1 Soundness of the Type System with Subsumption

The proof of the Type Soundness concerning the type assignment system with subsumption $\lambda\mathcal{O}bj^{\oplus}_{S}$ 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 proof of the crucial Theorem 6.9 needs to be modified significantly. Therefore, we do not document 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 is well-formed), 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 object-types), 6.5 (Matching), 6.6 (Match Weakening), as follows.

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

(ii) $\Gamma \vdash t \oplus \overline{m} : *$ if and only if Γ contains either $t \mathrel{\prec\!\!\!\#} \mathit{prot}\, t.R \oplus \overline{n}$ or $t \mathrel{\prec\!\!\!\#} \mathit{obj}\, t.R \oplus \overline{n}$, with $\overline{m} \subseteq \overline{R}$.

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

(vi) (Uniqueness) if $\Gamma \vdash \tau_1 \mathrel{\prec\!\!\!\#} \mathit{obj}\, t.\langle R_1, m{:}\sigma_1\rangle$ and $\Gamma \vdash \tau_1 \mathrel{\prec\!\!\!\#} \mathit{obj}\, t.\langle R_2, m{:}\sigma_2\rangle$, then $\sigma_1 \equiv \sigma_2$.

Moreover, in the same lemma the following points need to be added:

(i') $\Gamma \vdash \mathit{obj}\, t.R_1 \oplus \overline{m} \mathrel{\prec\!\!\!\#} \tau_2$ if and only if $\Gamma \vdash \mathit{prot}\, t.R_1 \oplus \overline{m} : *$ and $\Gamma \vdash \tau_2 : *$ and $\tau_2 \equiv \mathit{obj}\, t.R_2 \oplus \overline{n}$, with $R_2 \subseteq R_1$ and $\overline{n} \subseteq \overline{m}$.

(iii') $\Gamma \vdash t \oplus \overline{m} \mathrel{\prec\!\!\!\#} \mathit{obj}\, t.R_2 \oplus \overline{n}$ if and only if Γ contains either $t \mathrel{\prec\!\!\!\#} \mathit{obj}\, t.R_1 \oplus \overline{p}$ or $t \mathrel{\prec\!\!\!\#} \mathit{prot}\, t.R_1 \oplus \overline{p}$, with $R_2 \subseteq R_1$ and $\overline{n} \subseteq \overline{m} \cup \overline{p}$.

(viii') If $\Gamma \vdash \tau_1 \oplus m \mathrel{\prec\!\!\!\#} \mathit{obj}\, t.R \oplus \overline{n}$, then $\Gamma \vdash \tau_1 \mathrel{\prec\!\!\!\#} \mathit{obj}\, t.R \oplus \overline{n}{-}m$.

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

(ii) If $\Gamma \vdash \mathit{prot}\, t.\langle R, n{:}\sigma\rangle \oplus \overline{m} : *$ or $\Gamma \vdash \mathit{obj}\, t.\langle R, n{:}\sigma\rangle \oplus \overline{m} : *$ can be derived, then $\Gamma, t \mathrel{\prec\!\!\!\#} \mathit{obj}\, t.\langle R, n{:}\sigma\rangle \oplus \overline{m} \vdash \sigma : *$.

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 $\Gamma \vdash \tau_1 \mathrel{\prec\!\!\!\#} \tau_2$, then $\Gamma \vdash \sigma[\tau_1/t] \mathrel{\prec\!\!\!\#} \sigma[\tau_2/t]$.

(ii) If $\Gamma \vdash \sigma_1 : *_{rgd}$ and $\Gamma \vdash \sigma_2 : *_{rgd}$, then $\Gamma \vdash \sigma_1[\sigma_2/t] : *_{rgd}$.

Finally, the Subject Reduction for the type assignment system with subsumption has the usual formulation, but needs a more complex proof (reported in Appendix D).

Theorem 6.13 *(Subject Reduction, $\lambda\mathcal{O}bj^{\oplus}_{S}$) If $\Gamma \vdash e : \beta$ and $e \to e'$, then $\Gamma \vdash e' : \beta$.*

7 Object reclassification

The natural counterpart of self-extension in class-based languages is known as "(dynamic) object reclassification". This operation provides with the possibility of changing at runtime the class membership of an object while retaining its identity. 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 $\lambda\mathcal{O}bj^{\oplus}$.

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 Fickle$_{II}$ [DDDG02]).

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`) 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. We remark 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).

```
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 – Person-Student-Worker example

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);
```

Line 4, by sending the `registration` message to the object `q`, causes the execution of the reclassification expression `this=>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=>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 $\lambda\mathcal{O}bj^{\oplus}$ w.r.t. the reclassification goal, without guaranteeing that the terms we introduce can be typed.

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, as follows:

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

To model the example of Figure 3 in $\lambda\mathcal{O}bj^{\oplus}$, we have defined the *alice* object prototype 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. Therefore, $alice \Leftarrow reg(45)$ reduces to the following object:

$$\begin{array}{lcrcl}
alice_S & \triangleq & \langle name, reg, emp & = & \text{as in } alice, \\
 & & id & = & 45, \\
 & & emp & = & \lambda m.alice \Leftarrow emp(m)\rangle
\end{array}$$

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). In fact, $alice_S \Leftarrow emp(30K)$ reduces to:

$$\begin{array}{lcrcl}
alice_W & \triangleq & \langle name, reg, emp & = & \text{as in } alice, \\
 & & sal & = & 30K, \\
 & & reg & = & \lambda m.alice \Leftarrow reg(m), \\
 & & emp & = & \lambda s.\lambda n.\langle s \leftarrow\!\!\oplus\, sal = (s \Leftarrow sal) + n\rangle\rangle
\end{array}$$

As the reader can see, the effect of this message is that the methods characterizing 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 the modeling of Section's 7.1 example in our calculus, $alice_W$'s income may be increased by means of a call to such a version of *emp*, which has overridden

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

alice's third method; i.e. $alice_W \Leftarrow emp(14K)$ reduces to:

$$\begin{array}{rclrcl} alice_{W_2} & \triangleq & \langle name, reg, emp, sal, reg, emp & = & \text{as in } alice_W, \\ & & sal & = & (alice_W \Leftarrow sal) + 14K\rangle \end{array}$$

About tipability. The encoding devised in this section may be seen as a reasonable solution to emulate object reclassification in $\lambda\mathcal{O}bj^{\oplus}$; 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, representing the receiver object, 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*) object:

$$andback \triangleq \langle extend = \lambda s.\langle s \hookleftarrow\!\!\oplus\, delete = \lambda s'.s\rangle\rangle$$

The difficulty to type *andback* concerns the type returned by the *delete* method:

$$andback \;:\; pro\,t.\langle extend{:}t \oplus delete,\; delete{:}?\rangle \oplus extend$$

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 *andback* itself with the type returned by *delete*:

$$andback \;:\; \tau \triangleq pro\,t.\langle extend{:}t \oplus delete,\; delete{:}\tau\rangle \oplus extend$$

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

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 Figure 3's example via the following prototype:

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

As the reader can inspect, in this alternative Alice's encoding the variables representing the host object (s and s') are never used in a method body to represent the receiver 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:

$$\begin{array}{rcrcl} alice' & : & prot.\langle name & : & String, \\ & & reg & : & \mathbb{N} \to t \oplus id \oplus sal, \\ & & emp & : & \mathbb{N} \to t \oplus id \oplus sal, \\ & & id & : & \mathbb{N}, \\ & & sal & : & \mathbb{N}\rangle \oplus name, reg, emp \end{array} \tag{1}$$

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, i.e. $alice' \Leftarrow reg(45)$ reduces to:

$$\begin{array}{rcrcl} alice'_S & \triangleq & \langle name, reg, emp & = & \text{as in } alice', \\ & & id & = & 45, \\ & & sal & = & 0\rangle \end{array}$$

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 will 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].

We proceed by reclassifying $alice'_S$ into a worker; $alice'_S \Leftarrow emp(30K)$ reduces to[3]:

$$\begin{array}{rcrcl} alice'_W & \triangleq & \langle name, reg & = & \text{as in } alice', \\ & & id & = & 0, \\ & & sal & = & 30K, \\ & & emp & = & \lambda s.\lambda m.\langle\langle s \leftarrow\!\!\oplus\, id = 0\rangle \\ & & & & \qquad\qquad \leftarrow\!\!\oplus\, sal = (s \Leftarrow sal) + m\rangle\rangle \end{array}$$

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* is embedded in the interface. Notice that such a novel *emp* (incrementing the salary *sal*) works correctly not only with the usual multi-worker operation $alice'_W \Leftarrow emp(14K)$, reducing to:

$$\begin{array}{rcrcl} alice'_{W_2} & \triangleq & \langle name, reg, emp & = & \text{as in } alice'_W, \\ & & id & = & 0, \\ & & sal & = & (alice'_W \Leftarrow sal) + 14K\rangle \end{array}$$

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 (1). 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 $\mathbb{N} \to t \oplus id \oplus sal$. 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.

[3]Notice that, to ease readability, 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*).

7.4 Creating new objects

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

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

which models the same behavior of the minimal *andback*, 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 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):

$$andback' \;:\; \tau' \;\triangleq\; \mathit{prot}.\langle extend{:}\mathit{prot}'.\langle extend{:}t', delete{:}t\rangle \oplus extend, delete\rangle \oplus extend$$

The reader may observe how the type τ' 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:

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

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 belongs to a different object, created ex-novo. In the end, the $alice''$ prototype object can type-checked against the following type[4]:

$$\begin{array}{rcrcl} alice'' : \rho & \triangleq & \mathit{prot}.\langle name & : & String, \\ & & reg & : & \mathbb{N} \to \mathit{prot}'.\langle name : String, \\ & & & & \qquad id : \mathbb{N}, \\ & & & & \qquad emp : \mathbb{N} \to t \oplus sal\rangle \oplus name, id, emp, \\ & & emp & : & \mathbb{N} \to t \oplus sal, \\ & & sal & : & \mathbb{N}\rangle \oplus name, reg, emp \end{array}$$

where it is apparent that both the *emp* versions add *sal* to $alice''$'s interface. Then, the outcome of Alice's registration, $alice'' \Leftarrow reg(45)$, is the following:

$$\begin{array}{rcrcl} alice''_S & \triangleq & \langle name & = & alice'' \Leftarrow name, \\ & & id & = & 45, \\ & & emp & = & \lambda m.\langle alice'' \leftarrow\!\!\oplus\, sal = 0\rangle \Leftarrow emp(m)\rangle \end{array}$$

$$\begin{array}{rcrcl} alice''_S & : & \mathit{prot}'.\langle name & : & String, \\ & & id & : & \mathbb{N}, \\ & & emp & : & \mathbb{N} \to \rho \oplus sal\rangle \oplus name, id, emp \end{array}$$

[4]Typing the third method *emp* is not problematic, being simpler than in previous Section 7.3.

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 $alice''$, via the $alice''_S \Leftarrow emp(30K)$ call, is crucial for the reclassification, giving in fact the following outcome:

$$\begin{array}{rcrcl} alice''_W & \triangleq & \langle name, reg & = & \text{as in } alice'', \\ & & sal & = & 30K, \\ & & emp & = & \lambda s.\lambda m.\langle s \leftarrow\!\oplus\, sal = (s \Leftarrow sal) + m \rangle\rangle \end{array}$$

$$\begin{array}{rcrcl} alice''_W & : & prot.\langle name & : & String, \\ & & reg & : & \mathbb{N} \to prot'.\langle name : String, \\ & & & & \qquad\quad id : \mathbb{N}, \\ & & & & \qquad\quad emp : \mathbb{N} \to t\rangle \oplus name, id, emp \\ & & sal & : & \mathbb{N}, \\ & & emp & : & \mathbb{N} \to t\rangle \oplus name, reg, sal, emp \end{array}$$

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, through the $alice''_W \Leftarrow emp(14K)$ call, which reduces to the following object, whose type is the same of $alice''_W$:

$$\begin{array}{rcrcl} alice''_{W_2} & \triangleq & \langle name, reg, emp & = & \text{as in } alice''_W, \\ & & sal & = & (alice''_W \Leftarrow sal) + 14K\rangle \end{array}$$

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:

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

Summarizing, 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

Several efforts have been carried out in recent years with an aim similar to that of our work, namely 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 In prototype-based languages

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 [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 $\lambda\mathcal{O}bj^{\oplus}$ calculus, JavaScript provides self-inflicted extension; to deal with this feature, some ideas shared with our approach are adopted, namely *i*) the distinction between *pro*-types and *obj*-types, *ii*) the distinction between "available" and "reserved" methods, and *iii*) the mechanisms to mark the migration of a method from reserved to available. On the other hand, the main differences or extra features w.r.t. our work are the following: *i*) JavaScript allows strong update, i.e. overriding a method with a different type, and the type system accommodates, in a limited way, this functionality; *ii*) the types are defined by means of a set of subtyping constraints; *iii*) the syntax is completely different.

Chugh and co-workers propose in [CHJ12] a static type system for quite a rich subset of JavaScript. 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 authors further elaborate our idea of splitting the list of methods into reserved and available parts.

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

8.2 In class-based languages

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

$\lambda\mathcal{O}bj^{\oplus}$ the reclassification mechanism implemented in Fickle [DDDG02]. We complete now the survey of the involved related work by presenting other contributions that fall in the same class-based category.

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 $\lambda\mathcal{O}bj^{\oplus}$). 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.

Monpratarnchai and Tamai [MT08] introduce an extension of Java named EpsilonJ, featuring role modeling (that is, a set of roles to 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). Dynamically acquired methods obtained by assuming roles have to be invoked by means of down-casting, which is a type unsafe operation. Later, Kamina and Tamai [KT10] introduce 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.

Ressia and co-workers [RGN$^+$14] introduce a new form of inheritance called *talents*. 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).

8.3 Object extension vs. subsumption

Several calculi proposed in the literature combine object extension with object subsumption. Beside of the peculiar technicalities of those proposals, they all share the principle of avoiding (type incompatible) object extensions in presence of a (limited) form of object subsumption.

Riecke and Stone in [RS98] present 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 with the goal of linking correctly method names and method bodies.

Ghelli in [Ghe02] pursues the same freedom (of forgetting a method and adding it again with a different incompatible type) by introducing a context-dependent behaviour of objects called *object role*. Ghelli introduces a role calculus, which is a minimal extension of Abadi-Cardelli's ς-calculus, where an object is allowed to change dynamically identity while keeping static type checking. The “view" Vouillon’s mechanism [Vou01], see Section 8.1, can also be interpreted as a a kind of role.

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-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 would not be typable in that system:

$$\langle\langle p \leftarrow\!\!\oplus\, new_m = \ldots\rangle \Leftarrow add_{col}\rangle \Leftarrow new_m$$

$$\langle\langle p \Leftarrow add_{col}\langle\leftarrow\!\!\oplus\, new_m = \ldots\rangle$$

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*.

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.

[BBL96] V. Bono, M. Bugliesi, and L. Liquori. A Lambda Calculus of Incomplete Objects. In *Proc. of MFCS*, volume 1113, pages 218–229, 1996.

[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.

[Ghe02] Giorgio Ghelli. Foundations for extensible objects with roles. *Information and Computation*, 175(1):50–75, 2002.

[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.

[MT08] Supasit Monpratarnchai and Tetsuo Tamai. The implementation and execution framework of a role model based language, epsilonj. In *Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, SNPD 2008, In conjunction with Second International Workshop on Advanced Internet Technology and Applications, August 6-8, 2008, Phuket, Thailand*, pages 269–276, 2008. URL: `https://doi.org/10.1109/SNPD.2008.103`, `doi:10.1109/SNPD.2008.103`.

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

[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`.

A Typing rules, $\lambda\mathcal{O}bj^{\oplus}$

Well-formed Contexts

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

$$\frac{\Gamma \vdash \sigma : * \quad x \notin Dom(\Gamma)}{\Gamma, x{:}\sigma \vdash ok}\ (Cont{-}x)$$

$$\frac{\Gamma \vdash prot.R \oplus \overline{m} : * \quad t \notin Dom(\Gamma)}{\Gamma, t \prec\!\!\!\sharp\, prot.R \oplus \overline{m} \vdash ok}\ (Cont{-}t)$$

Well-formed Types

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

$$\frac{\Gamma \vdash \sigma_1 : * \quad \Gamma \vdash \sigma_2 : *}{\Gamma \vdash \sigma_1 \to \sigma_2 : *}\ (Type{-}Arrow)$$

$$\frac{\Gamma \vdash ok}{\Gamma \vdash prot.\langle\,\rangle : *}\ (Type{-}Pro_{\langle\,\rangle})$$

$$\frac{\Gamma, t \prec\!\!\!\sharp\, prot.R \vdash \sigma : * \quad m \notin \overline{R}}{\Gamma \vdash prot.\langle R, m{:}\sigma\rangle : *}\ (Type{-}Pro)$$

$$\frac{\Gamma \vdash \tau \prec\!\!\!\sharp\, prot.R \quad \overline{m} \subseteq \overline{R}}{\Gamma \vdash \tau \oplus \overline{m} : *}\ (Type{-}Extend)$$

Matching Rules

$$\frac{\Gamma \vdash t \oplus \overline{m} : * \quad \overline{n} \subseteq \overline{m}}{\Gamma \vdash t \oplus \overline{m} \prec\!\!\!\sharp\, t \oplus \overline{n}}\ (Match{-}t)$$

$$\frac{\Gamma \vdash prot.R_1 \oplus \overline{m} : * \quad \Gamma \vdash prot.R_2 \oplus \overline{n} : * \quad R_2 \subseteq R_1 \quad \overline{n} \subseteq \overline{m}}{\Gamma \vdash prot.R_1 \oplus \overline{m} \prec\!\!\!\sharp\, prot.R_2 \oplus \overline{n}}\ (Match{-}Pro)$$

$$\frac{\Gamma_1, t \prec\!\!\!\sharp\, \tau_1, \Gamma_2 \vdash \tau_1 \oplus \overline{m} \prec\!\!\!\sharp\, \tau_2}{\Gamma_1, t \prec\!\!\!\sharp\, \tau_1, \Gamma_2 \vdash t \oplus \overline{m} \prec\!\!\!\sharp\, \tau_2}\ (Match{-}Var)$$

Type Rules for λ-terms

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

$$\frac{\Gamma_1, x{:}\sigma, \Gamma_2 \vdash ok}{\Gamma_1, x{:}\sigma, \Gamma_2 \vdash x : \sigma}\ (Var)$$

$$\frac{\Gamma, x{:}\sigma_1 \vdash e : \sigma_2}{\Gamma \vdash \lambda x.e : \sigma_1 \rightarrow \sigma_2}\ (Abs)$$

$$\frac{\Gamma \vdash e_1 : \sigma_1 \rightarrow \sigma_2 \quad \Gamma \vdash e_2 : \sigma_1}{\Gamma \vdash e_1 e_2 : \sigma_2}\ (Appl)$$

Type Rules for Object Terms

$$\frac{\Gamma \vdash ok}{\Gamma \vdash \langle\,\rangle : prot.\langle\,\rangle}\ (Empty)$$

$$\frac{\Gamma \vdash e : prot.R_1 \oplus \overline{m} \quad \Gamma \vdash prot.\langle R_1, R_2\rangle \oplus \overline{m} : *}{\Gamma \vdash e : prot.\langle R_1, R_2\rangle \oplus \overline{m}}\ (Pre{-}Extend)$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash \tau \prec\!\!\!\sharp\, prot.\langle R, n{:}\sigma\rangle \oplus \overline{m} \\ \Gamma, t \prec\!\!\!\sharp\, prot.\langle R, n{:}\sigma\rangle \oplus \overline{m}, n \vdash e_2 : t \rightarrow \sigma\end{array}}{\Gamma \vdash \langle e_1 \leftarrow\!\!\oplus\, n = e_2\rangle : \tau \oplus n}\ (Extend)$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash \tau \prec\!\!\!\sharp\, prot.\langle R, n{:}\sigma\rangle \oplus \overline{m}, n \\ \Gamma, t \prec\!\!\!\sharp\, prot.\langle R, n{:}\sigma\rangle \oplus \overline{m}, n \vdash e_2 : t \rightarrow \sigma\end{array}}{\Gamma \vdash \langle e_1 \leftarrow\!\!\oplus\, n = e_2\rangle : \tau}\ (Override)$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau \prec\!\!\!\sharp\, prot.\langle R, n{:}\sigma\rangle \oplus \overline{m}, n}{\Gamma \vdash e \Leftarrow n : \sigma[\tau/t]}\ (Send)$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash \tau \prec\!\!\!\sharp\, prot.\langle R, n{:}\sigma\rangle \oplus \overline{m}, n \\ \Gamma, t \prec\!\!\!\sharp\, prot.\langle R, n{:}\sigma\rangle \oplus \overline{m}, n \vdash e_2 : t \rightarrow t \oplus \overline{n}\end{array}}{\Gamma \vdash Sel(e_1, n, e_2) : \sigma[\tau \oplus \overline{n}/t]}\ (Select)$$

B Extra rules for Subsumption, $\lambda\mathcal{O}bj^{\oplus}_{S}$

Extra Well-formed Contexts

$$\frac{\Gamma \vdash obj\, t.R \oplus \overline{m} : * \quad t \notin Dom(\Gamma)}{\Gamma, t \mathrel{\prec\!\!\!\#} obj\, t.R \oplus \overline{m} \vdash ok} \ (Cont\!-\!Obj)$$

Extra Well-formed Types

$$\frac{\Gamma \vdash prot\, t.R \oplus \overline{m} : *}{\Gamma \vdash obj\, t.R \oplus \overline{m} : *} \ (Type\!-\!Obj)$$

$$\frac{\Gamma \vdash \tau \mathrel{\prec\!\!\!\#} obj\, t.R \quad \overline{m} \subseteq \overline{R}}{\Gamma \vdash \tau \oplus \overline{m} : *} \ (Type\!-\!Extend\!-\!Obj)$$

Rules for Rigid Types

$$\frac{\Gamma \vdash ok}{\Gamma \vdash \iota : *_{rgd}} \ (Type\!-\!Const\!-\!Rgd)$$

$$\frac{\Gamma \vdash \sigma_1 : * \quad \Gamma \vdash \sigma_2 : *_{rgd}}{\Gamma \vdash \sigma_1 \to \sigma_2 : *_{rgd}} \ (Type\!-\!Arrow\!-\!Rgd)$$

$$\frac{\Gamma_1, t \mathrel{\prec\!\!\!\#} obj\, t.R \oplus \overline{m}, \Gamma_2 \vdash t \oplus \overline{n} : * \quad t \text{ covariant in } R}{\Gamma_1, t \mathrel{\prec\!\!\!\#} obj\, t.R \oplus \overline{m}, \Gamma_2 \vdash t \oplus \overline{n} : *_{rgd}} \ (Type\!-\!Var\!-\!Obj)$$

$$\frac{\Gamma \vdash obj\, t.\langle \overline{m}_k : \overline{\sigma}_k \rangle \oplus \overline{n} : * \quad \forall i \leq k.\ \Gamma \vdash \sigma_i : *_{rgd} \wedge t \text{ covariant in } \sigma_i}{\Gamma \vdash obj\, t.\langle \overline{m}_k : \overline{\sigma}_k \rangle \oplus \overline{n} : *_{rgd}} \ (Type\!-\!Obj\!-\!Rdg)$$

Extra Matching Rules

$$\frac{\Gamma \vdash \sigma_1' \mathrel{\prec\!\!\!\#} \sigma_1 \quad \Gamma \vdash \sigma_2 \mathrel{\prec\!\!\!\#} \sigma_2' \quad \Gamma \vdash \sigma_1 : *_{rgd}}{\Gamma \vdash \sigma_1 \to \sigma_2 \mathrel{\prec\!\!\!\#} \sigma_1' \to \sigma_2'} \ (Match\!-\!Arrow)$$

$$\frac{\Gamma \vdash prot\, t.R_1 \oplus \overline{m} : * \quad \Gamma \vdash prot\, t.R_2 \oplus \overline{n} : * \quad R_2 \subseteq R_1 \quad \overline{n} \subseteq \overline{m}}{\Gamma \vdash prot\, t.R_1 \oplus \overline{m} \mathrel{\prec\!\!\!\#} obj\, t.R_2 \oplus \overline{n}} \ (Promote)$$

$$\frac{\Gamma \vdash prot\, t.R_1 \oplus \overline{m} : * \quad \Gamma \vdash prot\, t.R_2 \oplus \overline{n} : * \quad R_2 \subseteq R_1 \quad \overline{n} \subseteq \overline{m}}{\Gamma \vdash obj\, t.R_1 \oplus \overline{m} \mathrel{\prec\!\!\!\#} obj\, t.R_2 \oplus \overline{n}} \ (Match\!-\!Obj)$$

Extra Type Rules for Terms

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash \tau \mathrel{\prec\!\!\!\#} obj\, t.\langle R, n{:}\sigma \rangle \oplus \overline{m} \\ \Gamma, t \mathrel{\prec\!\!\!\#} obj\, t.\langle R, n{:}\sigma \rangle \oplus \overline{m}, n \vdash e_2 : t \to \sigma\end{array}}{\Gamma \vdash \langle e_1 \leftarrow\!\!\!\oplus\, n = e_2 \rangle : \tau \oplus n} \ (Extend\!-\!Obj)$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash \tau \mathrel{-\!\!\!\sharp} obj\, t.\langle R, n{:}\sigma\rangle \oplus \overline{m}, n \qquad \Gamma, t \mathrel{-\!\!\!\sharp} obj\, t.\langle R, n{:}\sigma\rangle \oplus \overline{m}, n \vdash e_2 : t \to \sigma}{\Gamma \vdash \langle e_1 \leftarrow\!\!\oplus\, n = e_2\rangle : \tau} \; (Override{-}Obj)$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma \vdash \tau \mathrel{-\!\!\!\sharp} obj\, t.\langle R, n{:}\sigma\rangle \oplus \overline{m}, n}{\Gamma \vdash e \Leftarrow n : \sigma[\tau/t]} \; (Send{-}Obj)$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma \vdash \tau \mathrel{-\!\!\!\sharp} obj\, t.\langle R, n{:}\sigma\rangle \oplus \overline{m}, n \qquad \Gamma, t \mathrel{-\!\!\!\sharp} obj\, t.\langle R, n{:}\sigma\rangle \oplus \overline{m}, n \vdash e_2 : t \to t \oplus \overline{n}}{\Gamma \vdash Sel(e_1, n, e_2) : \sigma[\tau \oplus \overline{n}/t]} \; (Select{-}Obj)$$

$$\frac{\Gamma \vdash e : \sigma_1 \quad \Gamma \vdash \sigma_1 \mathrel{-\!\!\!\sharp} \sigma_2 \quad \Gamma \vdash \sigma_2 : *_{rgd}}{\Gamma \vdash e : \sigma_2} \; (Subsume)$$

C Soundness of the Type System $\lambda\mathcal{O}bj^{\oplus}$

Lemma C.1 *(Sub-derivation)*

(i) If Δ is a derivation of $\Gamma_1, \Gamma_2 \vdash \mathcal{A}$, then there exists a sub-derivation $\Delta' \subseteq \Delta$ of $\Gamma_1 \vdash ok$.

(ii) If Δ is a derivation of $\Gamma_1, x{:}\sigma, \Gamma_2 \vdash \mathcal{A}$, then there exists a sub-derivation $\Delta' \subseteq \Delta$ of $\Gamma_1 \vdash \sigma : *$.

(iii) If Δ is a derivation of $\Gamma_1, t \mathrel{-\!\!\!\sharp} \tau, \Gamma_2 \vdash \mathcal{A}$, then there exists a sub-derivation $\Delta' \subseteq \Delta$ of $\Gamma_1 \vdash \tau : *$.

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_2 is empty. In these cases the thesis coincides with the hypothesis. In 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 it is the case that Γ_2 is empty and the last rule in Δ is a context rule. In this latter case the last rule in Δ is necessarily a $(Cont{-}x)$ rule deriving $\Gamma_1, x{:}\sigma \vdash ok$, and the first premise of this rule coincides with the thesis.

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

Lemma C.2 *(Weakening)*

(i) If $\Gamma_1, \Gamma_2 \vdash \mathcal{A}$ and $\Gamma_1, \mathcal{C}, \Gamma_2 \vdash ok$, then $\Gamma_1, \mathcal{C}, \Gamma_2 \vdash \mathcal{A}$.

(ii) If $\Gamma_1 \vdash \mathcal{A}$ and $\Gamma_1, \Gamma_2 \vdash ok$, then $\Gamma_1, \Gamma_2 \vdash \mathcal{A}$.

(i) By structural induction on the derivation Δ of $\Gamma_1, \Gamma_2 \vdash \mathcal{A}$. If the last rule in Δ has the context in the conclusion identical to the context in the premise(s), then it is possible to apply the inductive hypothesis, thus deriving almost immediately the goal. In the other cases, if the last rule in Δ is a $(Cont{-}x)$ or $(Cont{-}t)$ rule, then the proof is trivial, since the second hypothesis coincides with the thesis. The remaining cases concern the $(Type{-}Pro)$, (Abs), $(Extend)$ and $(Override)$ rules, which require a more careful treatment. We detail here only the proof for $(Type{-}Pro)$, since the other rules are handled in a similar way.

In the $(Type{-}Pro)$ case, the hypothesis $\Gamma_1, \Gamma_2 \vdash prot.\langle R, m{:}\sigma\rangle : *$ follows from:

$$\Gamma_1, \Gamma_2, t \mathrel{-\!\!\!\!\lhd\!\sharp}\, prot.R \vdash \sigma : * \tag{2}$$

Let us briefly remark that if the statement $\mathcal{C}$ of the second hypothesis is equal to $t \mathrel{\lhd\!\sharp}\, \tau$, for some type τ, then it is convenient to α-convert the type $prot.\langle R, m{:}\sigma\rangle$ to avoid clash of variables. In any case, by Lemma C.1.(iii) (Sub-derivation), there exists a sub-derivation of Δ deriving $\Gamma_1, \Gamma_2 \vdash prot.R : *$, from which, by inductive hypothesis, $\Gamma_1, \mathcal{C}, \Gamma_2 \vdash prot.R : *$ and in turn, via the $(Cont{-}t)$ rule, $\Gamma_1, \mathcal{C}, \Gamma_2, t \mathrel{\lhd\!\sharp}\, prot.R \vdash ok$. By using (2) and the inductive hypothesis, we deduce $\Gamma_1, \mathcal{C}, \Gamma_2, t \mathrel{\lhd\!\sharp}\, prot.R \vdash \sigma : *$. Finally we have the thesis via the $(Type{-}Pro)$ rule.

(ii) By induction on the length of Γ_2; the proof uses the previous point (i) and Lemma C.1.(i) (Sub-derivation). $\square$

Lemma C.3 *(Well-formed object-types)*

(i) $\Gamma \vdash prot.R \oplus \overline{m} : *$ *if and only if* $\Gamma \vdash prot.R : *$ *and* $\overline{m} \subseteq \overline{R}$.

(ii) $\Gamma \vdash t \oplus \overline{m} : *$ *if and only if* Γ *contains* $t \mathrel{\lhd\!\sharp}\, prot.R \oplus \overline{n}$, *with* $\overline{m} \subseteq \overline{R}$.

Point (i) is immediately proved by inspection on the rules for well-formed types and matching. Point (ii) is proved by inspection on the rules for well-formed contexts, well-formed types and matching. $\square$

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

Proposition C.4 *(Matching is well-formed)*

If $\Gamma \vdash \tau_1 \mathrel{\lhd\!\sharp}\, \tau_2$, *then* $\Gamma \vdash \tau_1 : *$ *and* $\Gamma \vdash \tau_2 : *$.

By structural induction on the derivation Δ of $\Gamma \vdash \tau_1 \mathrel{\lhd\!\sharp}\, \tau_2$. The premises of the $(Match{-}Pro)$ rule coincide with the thesis. If the last rule in Δ is $(Match{-}t)$, we conclude by using its premises and Lemma C.3.(ii) (Well-formed object-types). If the last rule in Δ is $(Match{-}Var)$, then the judgment $\Gamma_1, t \mathrel{\lhd\!\sharp}\, \rho, \Gamma_2 \vdash t \oplus \overline{m} \mathrel{\lhd\!\sharp}\, \tau_2$ is derived from $\Gamma_1, t \mathrel{\lhd\!\sharp}\, \rho, \Gamma_2 \vdash \rho \oplus \overline{m} \mathrel{\lhd\!\sharp}\, \tau_2$. By inductive hypothesis τ_2 is well-formed and $\Gamma_1, t \mathrel{\lhd\!\sharp}\, \rho, \Gamma_2 \vdash \rho \oplus \overline{m} : *$. By inspecting the $(Cont{-}t)$ rule, ρ must be in the form $prot.R \oplus \overline{n}$, and by Lemma C.3.(i) (Well-formed types) it holds $\overline{m} \subseteq \overline{R}$. We can now conclude $\Gamma_1, t \mathrel{\lhd\!\sharp}\, \rho, \Gamma_2 \vdash t \oplus \overline{m} : *$ via Lemma C.3.(ii) (Well-formed object-types). $\square$

Lemma C.5 *(Matching)*

(i) $\Gamma \vdash prot.R_1 \oplus \overline{m} \mathrel{\lhd\!\sharp}\, \tau_2$ *if and only if* $\Gamma \vdash prot.R_1 \oplus \overline{m} : *$ *and* $\Gamma \vdash \tau_2 : *$ *and* $\tau_2 \equiv prot.R_2 \oplus \overline{n}$, *with* $R_2 \subseteq R_1$ *and* $\overline{n} \subseteq \overline{m}$.

(ii) $\Gamma \vdash \tau_1 \mathrel{\lhd\!\sharp}\, t \oplus \overline{n}$ *if and only if* $\Gamma \vdash \tau_1 : *$ *and* $\tau_1 \equiv t \oplus \overline{m}$, *with* $\overline{n} \subseteq \overline{m}$.

(iii) $\Gamma \vdash t \oplus \overline{m} \mathrel{\lhd\!\sharp}\, prot.R_2 \oplus \overline{n}$ *if and only if* Γ *contains* $t \mathrel{\lhd\!\sharp}\, prot.R_1 \oplus \overline{p}$, *with* $R_2 \subseteq R_1$ *and* $\overline{n} \subseteq \overline{m} \cup \overline{p}$.

(iv) *(Reflexivity) If* $\Gamma \vdash \rho : *$ *then* $\Gamma \vdash \rho \mathrel{\lhd\!\sharp}\, \rho$.

(v) (Transitivity) If $\Gamma \vdash \tau_1 \lhd\!\!\# \rho$ *and* $\Gamma \vdash \rho \lhd\!\!\# \tau_2$*, then* $\Gamma \vdash \tau_1 \lhd\!\!\# \tau_2$*.*

(vi) (Uniqueness) If $\Gamma \vdash \tau_1 \lhd\!\!\# prot.\langle R_1, m{:}\sigma_1\rangle$ *and* $\Gamma \vdash \tau_1 \lhd\!\!\# prot.\langle R_2, m{:}\sigma_2\rangle$*, then* $\sigma_1 \equiv \sigma_2$*.*

(vii) If $\Gamma \vdash \tau_1 \lhd\!\!\# \tau_2$ *and* $\Gamma \vdash \tau_2 \oplus m : *$*, then* $\Gamma \vdash \tau_1 \oplus m \lhd\!\!\# \tau_2 \oplus m$*.*

(viii) If $\Gamma \vdash \tau_1 \oplus m \lhd\!\!\# prot.R \oplus \overline{n}$*, then* $\Gamma \vdash \tau_1 \lhd\!\!\# prot.R \oplus \overline{n}{-}m$*.*

(ix) If $\Gamma \vdash \rho \oplus m : *$*, then* $\Gamma \vdash \rho \oplus m \lhd\!\!\# \rho$*.*

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

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

(v) By cases on the forms of τ_1, τ_2, ρ, using the points (i), (ii), (iii) above. If $\tau_1 \equiv prot.R \oplus \overline{m}$, we conclude by a triple application of point (i). If $\tau_2 \equiv t \oplus \overline{n}$, we conclude by three applications of point (ii). If $\tau_1 \equiv t \oplus \overline{m}$ and $\tau_2 \equiv prot.R \oplus \overline{n}$, we conclude by reasoning on the form of ρ, using all the 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 τ_1. If $\tau_1 \equiv prot.R \oplus \overline{m}$, we have the thesis by point (i) and Lemma C.3.(i) (Well-formed object-types). If $\tau_1 \equiv t \oplus \overline{m}$, we reason by cases on the form of τ_2: if $\tau_2 \equiv prot.R \oplus \overline{n}$, then we have the thesis by point (iii) and the validity of the thesis for `pro`-types; if $\tau_2 \equiv t \oplus \overline{n}$, then we have the thesis by point (ii).

(viii) By cases on the form of τ_1, using either point (i) or point (iii).

(ix) By cases on the form of ρ, using either point (i) or point (ii) and Lemma C.3.(ii) (Well-formed object-types). □

Lemma C.6 *(Match Weakening)*

(i) If $\Gamma_1, t \lhd\!\!\# \rho, \Gamma_2 \vdash \mathcal{A}$ *and* $\Gamma_1 \vdash \tau \lhd\!\!\# \rho$*, with* τ *a* `pro`*-type, then* $\Gamma_1, t \lhd\!\!\# \tau, \Gamma_2 \vdash \mathcal{A}$*.*

(ii) If $\Gamma \vdash prot.\langle R, n{:}\sigma\rangle \oplus \overline{m} : *$*, then* $\Gamma, t \lhd\!\!\# prot.\langle R, n{:}\sigma\rangle \oplus \overline{m} \vdash \sigma : *$*.*

(i) By structural induction on the derivation Δ of $\Gamma_1, t \lhd\!\!\# \rho, \Gamma_2 \vdash \mathcal{A}$.

The only case where the inductive hypothesis cannot be applied is when Γ_2 is empty and the last rule in Δ is a rule increasing the length of the context, i.e. the $(Cont{-}t)$ rule. In fact, $\Gamma, t \lhd\!\!\# \rho \vdash ok$ is derived from $t \notin Dom(\Gamma)$; on the other hand, from the second hypothesis and Lemma C.4 we have also that $\Gamma_1 \vdash \tau : *$, hence we may derive the thesis using the same $(Cont{-}t)$ 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 $t \lhd\!\!\# \rho$ in the context. The only rule that can use this hypothesis is $(Match{-}Var)$: in such a case $\Gamma_1, t \lhd\!\!\# \rho, \Gamma_2 \vdash t \oplus \overline{m} \lhd\!\!\# \upsilon$ is derived from the premise $\Gamma_1, t \lhd\!\!\# \rho, \Gamma_2 \vdash \rho \oplus \overline{m} \lhd\!\!\# \upsilon$. By inductive hypothesis, we have $\Gamma_1, t \lhd\!\!\# \tau, \Gamma_2 \vdash \rho \oplus \overline{m} \lhd\!\!\# \upsilon$. Moreover, from $\Gamma_1 \vdash \tau \lhd\!\!\# \rho$ and the Weakening Lemma C.2, we derive $\Gamma_1, t \lhd\!\!\# \tau, \Gamma_2 \vdash \tau \lhd\!\!\# \rho$, from which, by Lemma C.5.(vii), $\Gamma_1, t \lhd\!\!\# \tau, \Gamma_2 \vdash \tau \oplus \overline{m} \lhd\!\!\# \rho \oplus \overline{m}$. Finally, by transitivity of matching (Lemma C.5.(v)), we have $\Gamma_1, t \lhd\!\!\# \tau, \Gamma_2 \vdash \tau \oplus \overline{m} \lhd\!\!\# \upsilon$, and by an application of the $(Match{-}Var)$ rule we obtain the thesis.

(ii) First observe that there exists $R_1 \subseteq R$ such that $\Gamma, t \lhd\!\!\# prot.R_1 \vdash \sigma : *$.
In fact, by Lemma C.3.(i) (Well-formed object-types), we have $\Gamma \vdash prot.\langle R, n{:}\sigma\rangle : *$, that can only be derived by an application of the $(Type{-}Pro)$ rule; therefore, we have either our goal or $\Gamma, t \lhd\!\!\# prot.\langle R_2, n{:}\sigma\rangle \vdash \alpha : *$ for a suitable R_2 such that $R \equiv$

$\langle R_2, p{:}\alpha\rangle$. From Lemma C.1.(iii) (Sub-derivation) follows that $\Gamma \vdash prot.\langle R_2, n{:}\sigma\rangle : *$, hence we may conclude the existence of R_1.

Now, from $\Gamma, t \prec\!\!\sharp\, prot.R_1 \vdash \sigma : *$, by using Lemma C.1.(iii) (Sub-derivation), the $(Match{-}Pro)$ rule and point (i), we have the thesis. □

Proposition C.7 *(Substitution)*

(i) If $\Gamma_1, x{:}\sigma, \Gamma_2 \vdash \mathcal{A}$ *and* $\Gamma_1 \vdash e : \sigma$, *then* $\Gamma_1, \Gamma_2 \vdash \mathcal{A}[e/x]$.

(ii) If $\Gamma_1, t \prec\!\!\sharp\, \tau, \Gamma_2, \Gamma_3 \vdash \mathcal{A}$ *and* $\Gamma_1, t \prec\!\!\sharp\, \tau, \Gamma_2 \vdash \rho \prec\!\!\sharp\, \tau$, *then* $\Gamma_1, t \prec\!\!\sharp\, \tau, \Gamma_2, \Gamma_3[\rho/t] \vdash \mathcal{A}[\rho/t]$.

(iii) If $\Gamma_1, t \prec\!\!\sharp\, \tau, \Gamma_2 \vdash \mathcal{A}$ *and* $\Gamma_1 \vdash \rho \prec\!\!\sharp\, \tau$, *then* $\Gamma_1, \Gamma_2[\rho/t] \vdash \mathcal{A}[\rho/t]$.

(i) By induction on the derivation Δ of $\Gamma_1, x{:}\sigma, \Gamma_2 \vdash \mathcal{A}$. The only situation where the inductive hypothesis cannot be immediately applied is when the last rule in Δ is $(Cont{-}x)$. In such a case $\Gamma_1, x{:}\sigma \vdash ok$ is derived from $\Gamma_1 \vdash \sigma : *$, from which, by Lemma C.1.(i) (Sub-derivation), we have the thesis.

All the remaining rules can be easily managed by applying the inductive hypothesis, apart from the case 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 $\Gamma_1, x{:}\sigma, \Gamma_2 \vdash x : \sigma$ derives from the premise $\Gamma_1, x{:}\sigma, \Gamma_2 \vdash ok$ and so $\Gamma_1, \Gamma_2 \vdash ok$ by induction. By the second hypothesis $\Gamma_1 \vdash e : \sigma$ and Lemma C.2 (Weakening), we deduce $\Gamma_1, \Gamma_2 \vdash e : \sigma$.

(ii) By induction on the derivation Δ of $\Gamma_1, t \prec\!\!\sharp\, \tau, \Gamma_2, \Gamma_3 \vdash \mathcal{A}$. As in the previous point, the only case where the inductive hypothesis cannot be applied is when the last rule in Δ is a context rule; in this case the hypothesis coincides with the thesis.

About the remaining rules, the only non-trivial case is when the last rule in Δ is $(Match{-}Var)$ (the only rule that can use the judgment $t \prec\!\!\sharp\, \tau$ of the context) and the type variable t coincides with the one dealt with by the rule. In this case the conclusion $\Gamma_1, t \prec\!\!\sharp\, \tau, \Gamma_2, \Gamma_3 \vdash t \oplus \overline{m} \prec\!\!\sharp\, \tau_2$ derives from the premise $\Gamma_1, t \prec\!\!\sharp\, \tau, \Gamma_2, \Gamma_3 \vdash \tau \oplus \overline{m} \prec\!\!\sharp\, \tau_2$; then, by inductive hypothesis, $\Gamma_1, t \prec\!\!\sharp\, \tau, \Gamma_2, \Gamma_3[\rho/t] \vdash (\tau \oplus \overline{m} \prec\!\!\sharp\, \tau_2)[\rho/t]$. By the side condition on $(Cont{-}t)$, t cannot be free in τ and, by Lemma C.5 (i), neither in τ_2; hence, the above judgment can be written as $\Gamma_1, t \prec\!\!\sharp\, \tau, \Gamma_2, \Gamma_3[\rho/t] \vdash \tau \oplus \overline{m} \prec\!\!\sharp\, \tau_2$. On the other hand, from the second hypothesis $\Gamma_1, t \prec\!\!\sharp\, \tau, \Gamma_2 \vdash \rho \prec\!\!\sharp\, \tau$ we can derive $\Gamma_1, t \prec\!\!\sharp\, \tau, \Gamma_2, \Gamma_3[\rho/t] \vdash \rho \oplus \overline{m} \prec\!\!\sharp\, \tau \oplus \overline{m}$ by Lemma C.2.(ii) (Weakening) and Lemma C.5.(vii), and from the transitivity of matching (Lemma C.5.(v)) we can conclude $\Gamma_1, t \prec\!\!\sharp\, \tau, \Gamma_2, \Gamma_3[\rho/t] \vdash \rho \oplus \overline{m} \prec\!\!\sharp\, \tau_2$.

(iii) By the previous point we can derive $\Gamma_1, t \prec\!\!\sharp\, \tau, \Gamma_2[\rho/t] \vdash \mathcal{A}[\rho/t]$. Now, via an immediate induction, one can prove that if $\Gamma_1, t \prec\!\!\sharp\, \tau, \Gamma_2 \vdash \mathcal{A}$ and t is not free in Γ_2 nor in $\mathcal{A}$, then $\Gamma_1, \Gamma_2 \vdash \mathcal{A}$. The thesis follows immediately from such a property. □

Proposition C.8 *(Types of expressions are well-formed)*

If $\Gamma \vdash e : \beta$, *then* $\Gamma \vdash \beta : *$.

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

(Rules for λ-terms) If the last rule in Δ is $(Const)$, we derive the thesis via $(Type{-}Const)$. To address the (Var) rule we use Lemma C.1.(ii) (Sub-derivation) and Lemma C.2.(i) (Weakening). For the (Abs) rule one applies the inductive hypothesis, Lemma C.1.(ii) (Sub-derivation), Lemma C.7.(i) (Substitution), and the

($Type{-}Arrow$) rule. About ($Appl$), the inductive hypothesis allows us to derive $\Gamma \vdash \alpha{\to}\beta : *$; this judgment can only be derived through the ($Type{-}Arrow$) rule, whose second premise is precisely the thesis.

(Rules for object terms) The thesis is trivial for the ($Empty$), ($Pre{-}Extend$) and ($Override$) rules. In the ($Extend$) case, $\Gamma \vdash \langle e_1 \leftarrow\!\!\oplus\, n = e_2 \rangle : \tau \oplus n$ is derived from $\Gamma \vdash \tau \prec\!\!\sharp\, prot.\langle R, n{:}\sigma \rangle \oplus \overline{m}$; by Proposition C.4 and Lemma C.3.(i), we have $\Gamma \vdash prot.\langle R, n{:}\sigma \rangle \oplus \overline{m}, n : *$; by Lemma C.5.(vii), $\Gamma \vdash \tau \oplus n \prec\!\!\sharp\, prot.\langle R, n{:}\sigma \rangle \oplus \overline{m}, n$, and so we conclude by Proposition C.4. The two remaining cases are more complex.

($Send$) We have that $\Gamma \vdash e \Leftarrow n : \sigma[\tau/t]$ is derived from $\Gamma \vdash \tau \prec\!\!\sharp\, prot.\langle R, n{:}\sigma \rangle \oplus \overline{m}, n$, from which, by Proposition C.4, we derive $\Gamma \vdash prot.\langle R, n{:}\sigma \rangle \oplus \overline{m}, n : *$ and, in turn, $\Gamma, t \prec\!\!\sharp\, prot.\langle R, n{:}\sigma \rangle \oplus \overline{m}, n \vdash \sigma : *$ by Lemma C.6.(ii); finally, by Proposition C.7.(iii) (Substitution), we can conclude that $\Gamma \vdash \sigma[\tau/t] : *$.

($Select$) We have that $\Gamma \vdash Sel(e_1, n, e_2) : \sigma[(\tau \oplus \overline{n})/t]$ is derived from both $\Gamma, t \prec\!\!\sharp\, prot.\langle R, n{:}\sigma \rangle \oplus \overline{m}, n \vdash e_2 : t{\to}(t \oplus \overline{n})$ and $\Gamma \vdash \tau \prec\!\!\sharp\, prot.\langle R, n{:}\sigma \rangle \oplus \overline{m}, n$. By inductive hypothesis, $\Gamma, t \prec\!\!\sharp\, prot.\langle R, n{:}\sigma \rangle \oplus \overline{m}, n \vdash t{\to}(t \oplus \overline{n}) : *$ and, by Proposition C.7.(iii) (Substitution), $\Gamma \vdash \tau{\to}(\tau \oplus \overline{n}) : *$; then, since this latter judgment can only be obtained via the ($Type{-}Arrow$) rule, we deduce $\Gamma \vdash \tau \oplus \overline{n} : *$. Further, we have $\Gamma \vdash \tau \oplus \overline{n} \prec\!\!\sharp\, prot.\langle R, n{:}\sigma \rangle \oplus \overline{m}, n$ by case analysis and Lemma C.5.(i)-(iii), from which the thesis by Lemma C.6.(ii) and Proposition C.7.(iii) (Substitution). ☐

Theorem C.9 *(Subject Reduction, $\lambda\mathcal{O}bj^{\oplus}$) If $\Gamma \vdash e : \beta$ and $e \to e'$, then $\Gamma \vdash e' : \beta$.*

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

($Beta$) The derivation Δ of $\Gamma \vdash (\lambda x.e_1)e_2 : \beta$ needs to terminate with a rule ($Appl$), deriving $\Gamma \vdash (\lambda x.e_1)e_2 : \alpha$, potentially followed by some applications of ($Pre{-}Extend$). Let the premises of ($Appl$) be $\Gamma \vdash (\lambda x.e_1) : \sigma{\to}\alpha$ and $\Gamma \vdash e_2 : \sigma$ for a suitable σ; in turn, the first judgment has to be derived from $\Gamma, x{:}\sigma \vdash e_1 : \alpha$ via the (Abs) rule. By Proposition C.7.(i) (Substitution), we conclude $\Gamma \vdash (e_1 : \alpha)[e_2/x] \equiv e_1[e_2/x] : \alpha$; then, by repeating the potential applications of ($Pre{-}Extend$) in Δ, we have the thesis.

($Selection$) The derivation Δ of $\Gamma \vdash e \Leftarrow n : \beta$ has to terminate with a ($Send$) rule, deriving $\Gamma \vdash e \Leftarrow n : \sigma[\tau/t]$, potentially followed by applications of ($Pre{-}Extend$). The premises of ($Send$) are $\Gamma \vdash e : \tau$ and $\Gamma \vdash \tau \prec\!\!\sharp\, prot.\langle R, n{:}\sigma \rangle \oplus \overline{m}, n$. From this latter judgment, by Lemma C.4 (Matching is well-formed) and the rules ($Cont{-}t$), ($Match{-}Pro$), ($Match{-}Var$), ($Type{-}Extend$), ($Cont{-}x$), (Var), and (Abs), one can derive $\Gamma, t \prec\!\!\sharp\, prot.\langle R, n{:}\sigma \rangle \oplus \overline{m}, n \vdash \lambda s.s : t{\to}t$. From the above premises, by applying the ($Select$) rule, we have $\Gamma \vdash Sel(e, n, \lambda s.s) : \sigma[\tau/t]$ and, by repeating the potential applications of ($Pre{-}Extend$) in Δ, the thesis.

($Success$) The derivation Δ of $\Gamma \vdash Sel(\langle e_1 \leftarrow\!\!\oplus\, n = e_2 \rangle, n, e_3) : \beta$ must terminate with a ($Select$) rule, deriving $\Gamma \vdash Sel(\langle e_1 \leftarrow\!\!\oplus\, n = e_2 \rangle, n, e_3) : \sigma[(\tau \oplus \overline{n})/t]$, potentially followed by applications of ($Pre{-}Extend$). The premises of ($Select$) are:

$$\Gamma \vdash \langle e_1 \leftarrow\!\!\oplus\, n = e_2 \rangle : \tau \tag{3}$$

$$\Gamma \vdash \tau \prec\!\!\sharp\, prot.\langle R, n{:}\sigma \rangle \oplus \overline{m}, n \tag{4}$$

$$\Gamma, t \prec\!\!\sharp\, prot.\langle R, n{:}\sigma \rangle \oplus \overline{m}, n \vdash e_3 : t{\to}t \oplus \overline{n} \tag{5}$$

From (4) and (5), through the Substitution Lemma, we have $\Gamma \vdash e_3 : \tau{\rightarrow}\tau \oplus \overline{n}$; from this latter judgment and (3), by the (*Appl*) rule, we derive:

$$\Gamma \vdash e_3\langle e_1{\leftarrow\!\!\!\oplus}\, n = e_2\rangle : \tau \oplus \overline{n} \tag{6}$$

The judgment (3) can only be obtained using either the ($Extend$) rule or the ($Override$) one, potentially followed by some applications of ($Pre{-}Extend$). Here we consider only the case where ($Extend$) is applied, since ($Override$) can be managed similarly, with the difference that in some points the proof is simpler. Hence, let us assume that ($Extend$) derives $\Gamma \vdash \langle e_1{\leftarrow\!\!\!\oplus}\, n = e_2\rangle : \rho \oplus n$ from the premise $\Gamma \vdash e_1 : \rho$ and:

$$\Gamma \vdash \rho \mathrel{\prec\!\!\!\sharp} prot.\langle R_1, n{:}\sigma_1\rangle \oplus \overline{p} \tag{7}$$

$$\Gamma, t \mathrel{\prec\!\!\!\sharp} prot.\langle R_1, n{:}\sigma_1\rangle \oplus \overline{p}, n \vdash e_2 : t \rightarrow \sigma_1 \tag{8}$$

By inspection of the ($Pre{-}Extend$) rule, we can readily derive $\Gamma \vdash \tau \mathrel{\prec\!\!\!\sharp} \rho \oplus n$. From (7), by Lemma C.5.(vii), we have $\Gamma \vdash \rho{\oplus}n \mathrel{\prec\!\!\!\sharp} prot.\langle R_1, n{:}\sigma_1\rangle{\oplus}\overline{p}, n$, and, by transitivity of matching, $\Gamma \vdash \tau \mathrel{\prec\!\!\!\sharp} prot.\langle R_1, n{:}\sigma_1\rangle \oplus \overline{p}, n$. From this latter judgment and (4), by Lemma C.5.(vi) (Matching uniqueness), it follows that $\sigma \equiv \sigma_1$.

On the other hand, by Lemma C.5.(ix), we have $\Gamma \vdash \tau \oplus \overline{n} \mathrel{\prec\!\!\!\sharp} \tau$ and, by transitivity of matching, $\Gamma \vdash \tau \oplus \overline{n} \mathrel{\prec\!\!\!\sharp} prot.\langle R_1, n{:}\sigma\rangle \oplus \overline{p}, n$. From this latter judgment and (8), by the Substitution Lemma, we have $\Gamma \vdash e_2 : \tau \oplus \overline{n} \rightarrow \sigma[(\tau \oplus \overline{n})/t]$, and, in turn, from this and (6), $\Gamma \vdash e_2(e_3\langle e_1{\leftarrow\!\!\!\oplus}\, n = e_2\rangle) : \sigma[(\tau \oplus \overline{n})/t]$ via the (*Appl*) rule. Finally, by repeating the potential applications of ($Pre{-}Extend$) in Δ, we obtain the thesis.

($Next$) As argued for ($Success$), the derivation of $\Gamma \vdash Sel(\langle e_1{\leftarrow\!\!\!\oplus}\, n = e_2\rangle, m, e_3) : \beta$ must end with a ($Select$) rule, deriving $\Gamma \vdash Sel(\langle e_1{\leftarrow\!\!\!\oplus}\, n = e_2\rangle, m, e_3) : \sigma[(\tau \oplus \overline{m})/t]$, potentially followed by applications of ($Pre{-}Extend$). The premises of ($Select$) are:

$$\Gamma \vdash \langle e_1{\leftarrow\!\!\!\oplus}\, n = e_2\rangle : \tau \tag{9}$$

$$\Gamma \vdash \tau \mathrel{\prec\!\!\!\sharp} prot.\langle R, m{:}\sigma\rangle \oplus \overline{n}, m \tag{10}$$

$$\Gamma, t \mathrel{\prec\!\!\!\sharp} prot.\langle R, m{:}\sigma\rangle \oplus \overline{n}, m \vdash e_3 : t{\rightarrow}(t \oplus \overline{m}) \tag{11}$$

The judgment (9) can only be derived using either the ($Extend$) rule or the ($Override$) one, potentially followed by some applications of ($Pre{-}Extend$). As carried out in the proof for the ($Success$) rule, we address here only the case where ($Extend$) is applied, being the ($Override$) case similar but simpler.

Since ($Pre{-}Extend$) has been applied and (9) holds, τ must be in the form $prot.\langle R_1, m{:}\sigma, n{:}\sigma_1\rangle{\oplus}\overline{n}, m, n$. Hence, let (9) be derived through ($Pre{-}Extend$) from:

$$\Gamma \vdash \langle e_1{\leftarrow\!\!\!\oplus}\, n = e_2\rangle : prot.\langle R_2, m{:}\sigma, n{:}\sigma_1\rangle \oplus \overline{n}, m, n$$

(where $R_2 \subseteq R_1$), which, in turn, is derived via the ($Extend$) rule from the premises:

$$\Gamma \vdash e_1 : prot.\langle R_2, m{:}\sigma, n{:}\sigma_1\rangle \oplus \overline{n}, m \tag{12}$$

$$\Gamma \vdash prot.\langle R_2, m{:}\sigma, n{:}\sigma_1\rangle \oplus \overline{n}, m \mathrel{\prec\!\!\!\sharp} prot.\langle R_3, n{:}\sigma_1\rangle \oplus \overline{p} \tag{13}$$

$$\Gamma \vdash t \mathrel{\prec\!\!\!\sharp} prot.\langle R_3, n{:}\sigma_1\rangle \oplus \overline{p}, n \vdash e_2 : t \rightarrow \sigma_1 \tag{14}$$

Then, let ρ represent the type $prot.\langle R_1, m{:}\sigma, n{:}\sigma_1\rangle \oplus \overline{n}, m$, i.e. $\tau \equiv \rho \oplus n$. From the judgment (12), by the ($Pre{-}Extend$) rule, we can derive:

$$\Gamma \vdash e_1 : \rho \tag{15}$$

By the $(Match{-}Pro)$ rule, we have $\Gamma \vdash \rho \oplus n \mathrel{-\!\!\!\sharp} prot.\langle R_2, m{:}\sigma, n{:}\sigma_1\rangle \oplus \overline{n}, m$ and, from this latter judgment, (13) and (14), by transitivity of matching and the Weakening Lemma, we derive $\Gamma, t \mathrel{-\!\!\!\sharp} \rho \oplus n \vdash e_2 : t \to \sigma_1$. From it, by means of the (*Extend*) rule:

$$\Gamma, t \mathrel{-\!\!\!\sharp} \rho, s{:}t \vdash \langle s \leftarrow\!\!\oplus\, n = e_2\rangle : t \oplus n \tag{16}$$

Now, through (10), the $(Match{-}Var)$ rule, and the transitivity of matching, one can derive $\Gamma, t \mathrel{-\!\!\!\sharp} \rho \vdash t \oplus n \mathrel{-\!\!\!\sharp} prot.\langle R, m{:}\sigma\rangle \oplus \overline{n}, m$. From this latter judgment and (11), by Substitution, we obtain $\Gamma, t \mathrel{-\!\!\!\sharp} \rho \vdash e_3 : t \oplus n \to t \oplus n \oplus \overline{m}$, and, from this judgment and (16), by the (*Appl*) and (*Abs*) rules, we have:

$$\Gamma, t \mathrel{-\!\!\!\sharp} \rho \vdash \lambda s.e_3\langle s \leftarrow\!\!\oplus\, n = e_2\rangle : t \to t \oplus n \oplus \overline{m}$$

This judgment, together with (15), allows to apply the (*Select*) rule, thus deriving:

$$\Gamma \vdash Sel(e_1, m, \lambda s.e_3\langle s \leftarrow\!\!\oplus\, n = e_2\rangle) : \sigma[(\rho \oplus n \oplus \overline{m})/t]$$

Finally, we get the thesis via the usual potential applications of $(Pre{-}Extend)$. □

D Soundness of the Type System with Subsumption $\lambda\mathcal{O}bj^{\oplus}_{S}$

Theorem D.1 *(Subject Reduction, $\lambda\mathcal{O}bj^{\oplus}_{S}$) If $\Gamma \vdash e : \beta$ and $e \to e'$, then $\Gamma \vdash e' : \beta$.*

As in Theorem C.9, we prove that the type is preserved by each of the reduction rules (*Beta*), (*Selection*), (*Success*) and (*Next*). In the present case we have to manage the extra difficulty of potential applications of the (*Subsume*) rule.

(*Beta*) The derivation of $\Gamma \vdash (\lambda x.e_1)e_2 : \beta$ needs to terminate with a rule (*Appl*), deriving $\Gamma \vdash (\lambda x.e_1)e_2 : \alpha$, potentially followed by some applications of $(Pre{-}Extend)$ and (*Subsume*). The premises of (*Appl*) must be $\Gamma \vdash (\lambda x.e_1) : \sigma \to \alpha$ and $\Gamma \vdash e_2 : \sigma$, where the first judgment has to be derived via (*Abs*), followed by potential applications of (*Subsume*). Let $\Gamma \vdash (\lambda x.e_1) : \sigma_1 \to \alpha_1$ be the conclusion of the (*Abs*) rule, and:

$$\Gamma, x{:}\sigma_1 \vdash e_1 : \alpha_1 \tag{17}$$

its premise. Since the (*Subsume*) rule has been applied, we have $\Gamma \vdash \sigma_1 \to \alpha_1 \mathrel{-\!\!\!\sharp} \sigma \to \alpha$ and $\Gamma \vdash \sigma \to \alpha : *_{rgd}$, therefore $\Gamma \vdash \sigma \mathrel{-\!\!\!\sharp} \sigma_1$ and $\Gamma \vdash \sigma_1 : *_{rgd}$ and $\Gamma \vdash \alpha_1 \mathrel{-\!\!\!\sharp} \alpha$, where $\Gamma \vdash \alpha : *_{rgd}$. Using these judgments and (17) it is not difficult to prove, by structural induction, that $\Gamma, x{:}\sigma \vdash e_1 : \alpha_1$. By Substitution Lemma, we have then $\Gamma \vdash e_1[e_2/x] : \alpha_1$, and, by the (*Subsume*) rule, $\Gamma \vdash e_1[e_2/x] : \alpha$, from which the thesis.

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

(*Success*) As in Theorem C.9 (type system without subsumption), we can start by asserting that the derivation Δ of $\Gamma \vdash Sel(\langle e_1 \leftarrow\!\!\oplus\, n = e_2\rangle, n, e_3) : \beta$ must end with a (*Select*) rule, deriving $\Gamma \vdash Sel(\langle e_1 \leftarrow\!\!\oplus\, n = e_2\rangle, n, e_3) : \sigma[(\tau \oplus \overline{n})/t]$. This is potentially followed by applications of the $(Pre{-}Extend)$ rule and, in the present case, also the (*Subsume*) rule. The premises of (*Select*) are the following:

$$\Gamma \vdash \langle e_1 \leftarrow\!\!\oplus\, n = e_2\rangle : \tau \tag{18}$$

$$\Gamma \vdash \tau \mathrel{-\!\!\!\sharp} obj\, t.\langle R, n{:}\sigma\rangle \oplus \overline{m}, n \tag{19}$$

$$\Gamma, t \mathrel{\prec\!\!\!\sharp} obj\, t.\langle R, n{:}\sigma\rangle \oplus \overline{m}, n \vdash e_3 : t \rightarrow t \oplus \overline{n} \tag{20}$$

If the judgment (18) was not obtained by an application of the (*Subsume*) rule, we could repeat the steps argued to prove Theorem C.9. In fact, we address here the case where (18) is derived by a single application of (*Subsume*) (it sufficient to consider a single application, because consecutive applications can be always compacted into a single one). Hence, let the premises of (*Subsume*) be:

$$\Gamma \vdash \langle e_1 \leftarrow\!\!\!\oplus\, n = e_2\rangle : \rho \tag{21}$$

$$\Gamma \vdash \rho \mathrel{\prec\!\!\!\sharp} \tau \tag{22}$$

$$\Gamma \vdash \tau : *_{rgd} \tag{23}$$

From the judgments (19), (22) and (20), by transitivity of matching and Substitution, we have $\Gamma \vdash e_2 : \rho \rightarrow \rho \oplus \overline{n}$. From this and (21), by the (*Appl*) rule, we derive:

$$\Gamma \vdash e_3\langle e_1 \leftarrow\!\!\!\oplus\, n = e_2\rangle : \rho \oplus \overline{n} \tag{24}$$

Again, by repeating the steps carried out for Theorem C.9 (case analysis on the derivation of (21)), we can prove that $\Gamma \vdash e_2(e_3\langle e_1 \leftarrow\!\!\!\oplus\, n = e_2\rangle) : \sigma[(\rho \oplus \overline{n})/t]$.

Now, from (19) and (23) follows that t is covariant in σ and $\Gamma \vdash \sigma : *_{rgd}$, and from Lemma 6.12 that $\Gamma \vdash \sigma[(\rho \oplus \overline{n})/t] \mathrel{\prec\!\!\!\sharp} \sigma[(\tau \oplus \overline{n})/t]$ and $\Gamma \vdash \sigma[(\tau \oplus \overline{n})/t] : *_{rgd}$. Finally, by an application of the (*Subsume*) rule, we have $\Gamma \vdash e_2(e_3\langle e_1 \leftarrow\!\!\!\oplus\, m = e_2\rangle) : \sigma[(\tau \oplus \overline{n})/t]$, and from this the thesis via the applications of (*Pre−Extend*) potentially in Δ.

(*Next*) As in the version without subsumption, we start from the derivation Δ of $\Gamma \vdash Sel(\langle e_1 \leftarrow\!\!\!\oplus\, n = e_2\rangle, m, e_3) : \beta$, which has to terminate with a (*Select*) rule, deriving $\Gamma \vdash Sel(\langle e_1 \leftarrow\!\!\!\oplus\, n = e_2\rangle, m, e) : \sigma[(\tau \oplus \overline{m})/t]$, potentially followed by applications of the (*Pre−Extend*) and (*Subsume*) rules. Let the premises of (*Select*) be:

$$\Gamma \vdash \langle e_1 \leftarrow\!\!\!\oplus\, n = e_2\rangle : \tau \tag{25}$$

$$\Gamma \vdash \tau \mathrel{\prec\!\!\!\sharp} obj\, t.\langle R, m{:}\sigma\rangle \oplus \overline{n}, m \tag{26}$$

$$\Gamma, t \mathrel{\prec\!\!\!\sharp} obj\, t.\langle R, m{:}\sigma\rangle \oplus \overline{n}, m \vdash e_3 : t \rightarrow (t \oplus \overline{m}) \tag{27}$$

If the judgment (25) was not obtained by an application of the (*Subsume*) rule, we could repeat the steps argued to prove Theorem C.9. Then, we address here the case where (25) is derived by a single application of (*Subsume*), from the premises:

$$\Gamma \vdash \langle e_1 \leftarrow\!\!\!\oplus\, n = e_2\rangle : \rho \tag{28}$$

$$\Gamma \vdash \rho \mathrel{\prec\!\!\!\sharp} \tau \tag{29}$$

$$\Gamma \vdash \tau : *_{rgd} \tag{30}$$

From these hypotheses, by repeating the same steps argued for the proof without subsumption (case analysis on the derivation of the judgment (28)), we deduce:

$$\Gamma \vdash Sel(e_1, m, \lambda s.e_3\langle s \leftarrow\!\!\!\oplus\, n = e_2\rangle) : \sigma[(\rho \oplus n \oplus \overline{m})/t]$$

Finally, the proof can be accomplished as in the (*Success*) case, by applying Lemma 6.12 and by means of the (*Subsume*) and (*Pre−Extend*) rules. □

About the authors

Acknowledgments ACKS