A Lambda Calculus of Objects with Self-Inflicted Extension

Pietro Di Gianantonio
Dip. di Matematica ed Informatica
Università di Udine
I-33100 Udine, Italy
digianantonio@dimi.uniud.it

Furio Honsell
Dip. di Matematica ed Informatica
Università di Udine
I-33100 Udine, Italy
honsell@dimi.uniud.it

Luigi Liquori
Dip. di Matematica ed Informatica
Università di Udine
I-33100 Udine, Italy
liquori@dimi.uniud.it

Abstract

In this paper we investigate, in the context of *functional prototype-based languages*, objects which might extend themselves upon receiving a message. The possibility for an object of extending its own "self", referred to by Cardelli, as a *self-inflicted* operation, is novel in the context of typed object-based languages. We present a sound type system for this calculus which guarantees that evaluating a well-typed expression will never yield a `message-not-found` run-time error. We give several examples which illustrate the increased expressive power of our system with respect to existing calculi of objects. The new type system allows also for a flexible *width-subtyping*, still permitting sound method override, and a limited form of object extension. The resulting calculus appears to be a good starting point for a rigorous mathematical analysis of class-based languages.

1 Introduction

An untyped lambda calculus extended with object primitives, together with a sound type assignment system, were introduced by Fisher, Honsell, and Mitchell [16], as a solid foundation for functional typed object-oriented languages. In this system, called the *Lambda Calculus of Objects* ($\lambda\mathcal{O}bj$), objects are untyped, and a new object may be created by modifying or extending an existing *prototype*. The new object thereby inherits properties from the original one in a controlled manner. Objects can be viewed as lists of pairs *(method name, method body)* where the method body is (or reduces to) a lambda abstraction whose first formal parameter is always `self` (or `this` in C^{++}, $Java$).

The type assignment system of $\lambda\mathcal{O}bj$ is set up so as to prevent the unfortunate `message-not-found` run-time error. Types of methods are allowed to be *specialized* to the type of the inheriting objects. This feature, usually referred to as "Mytype method specialization", reinterprets the symbol `self` in the type of the inheriting object. This high mutability of method bodies is accommodated in the type system via an implicit form of *higher-order polymorphism*, inspired by the the work of Wand on extensible records [25].

In Proc. of ACM-SIGPLAN OOPSLA-98, International Symposium on Object Oriented, Programming, System, Languages and Applications, Vancouver, British Columbia, ACM Press, 1998.

The calculus $\lambda\mathcal{O}bj$ has spurred an intense research in type assignment systems for object calculi. Several calculi inspired by $\lambda\mathcal{O}bj$, which accommodate various extra features such as incomplete objects, subtyping, encapsulation, imperative features, have appeared in the literature in recent years (see *e.g.* [18, 7, 15, 5, 4, 19, 6]).

Independently, Abadi and Cardelli have introduced, with the same foundational spirit, the *Object Calculus* [1]. This is a calculus of *typed objects* and allows for a natural treatment of subtyping and hence of code reuse.

The essential differences between $\lambda\mathcal{O}bj$ and the Object Calculus of Abadi and Cardelli, are that objects in the latter calculus are *typed* and have a *fixed size*, while objects, in $\lambda\mathcal{O}bj$, are *untyped*, and can be *extended*, thereby changing dynamically their shape.

More specifically, $\lambda\mathcal{O}bj$ supports two operations which may change the shape of an object: *method addition* and *method override*. One of the most powerful features of the typing system is to allow for the typing of method bodies in objects which can modify their own `self`. Cardelli referred to this capability of a method of operating directly on its own self as a *self-inflicted* operation [12].

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

$$\begin{array}{lcl} \mathtt{point} \triangleq \langle \mathtt{x} & = & \lambda\mathtt{self}.1, \\ \quad\quad\quad\; \mathtt{set\_x} & = & \lambda\mathtt{self}.\lambda\mathtt{v}.\langle \mathtt{self} \longleftarrow \mathtt{x} = \lambda\mathtt{self}.\mathtt{v}\rangle, \\ \quad\quad\quad\; \ldots\rangle. & & \end{array}$$

When we send `set_x` *to* `point` *with an argument* `3`*, written as* $\mathtt{point} \Leftarrow \mathtt{set\_x}(3)$*, then we produce as a result 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. $\longleftarrow$*) performed by the method* `set_x`*.*

However, in all the type systems for lambda calculi of objects, both those derived from $\lambda\mathcal{O}bj$, and those derived from the Object Calculus, 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:

- in typed class-based languages we can modify the structure of the class only statically. If we need to add a new method to an instance of a class we are forced to *re-compile* the class and to make the modification needlessly available to *all the class-instances*, thereby wasting memory. If a class had a self-extension method, only the instances of the class which have dynamically executed this method would allocate new memory without the need of any re-compilation.

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

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

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

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

- an alternative principled explanation of the SmallTalk-80 class methods `addinstancevariable` and `addclassvariable` could be given naturally using self-extension;

- 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 he 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;

The objective of this paper is that of introducing $\lambda\mathcal{O}bj+$, a lambda calculus of objects in the style of $\lambda\mathcal{O}bj$, together with a type assignment system. The type assignment system allows self-inflicted object extension still catching statically the `message-not-found` run-time error. This system can be further extended to accommodate other "subtyping" features. By way of example we present a width-subtyping relation, that permits sound method override and a limited form of object extension.

Self-inflicted Extension.

To enable the $\lambda\mathcal{O}bj+$ calculus to perform self-inflicted extensions, two modifications of the system in [16] are necessary. The first is, in effect, a simplification of the original syntax of the language. The second is much more substantial and it involves the type discipline.

As far as the syntax of the language is concerned, we are forced to *unify* into a *single* operator, denoted by $\prec\!\!-$, the two original object operators of $\lambda\mathcal{O}bj$, *i.e.* object extension ($\leftarrow\!\!+$) and object override ($\longleftarrow$). 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, the second time we have just a simple object override.

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

$$\texttt{point} \;\stackrel{\triangle}{=}\; \langle \texttt{add\_set\_col} \;=\; \lambda\texttt{self}.\lambda\texttt{v}.\ \langle \texttt{self} \prec\!\!-\ \texttt{col} = \texttt{v}\rangle, \ \ldots\rangle.$$

When we send `add_set_col` *to* `point` *with argument* `white`, i.e. `point` $\Leftarrow$ `add_set_col (white)`*, then we produce as a result a new object* `col_point` *where the* `col` *field has been added to* `point` *and set to* `white`*. If we send twice* `add_set_col` *to* `point` *with an argument* `black`, i.e.

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

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

As far as types are concerned, we add two new kinds of types, namely $\tau \prec\!\!\!\leftarrow \texttt{m}$, which can be seen as the semantic counterpart of the syntactic $\langle e_1 \prec\!\!- \texttt{m} = e_2\rangle$ one, and $\texttt{pro}\,t.\langle\!\langle R \triangleleft R'\rangle\!\rangle$, which is a generalization of the original $\texttt{class}\,t.R$ in [16] (R is a *row* which contains all methods and related types present in the object-type $\texttt{class}\,t.R$).

If the type $\texttt{pro}\,t.\langle\!\langle R \triangleleft R'\rangle\!\rangle$ is assigned to an object e, then e can respond to all the methods listed in R. The list R', sometimes referred to as the "reservation" part of the object-type, represents the methods that can be added to e either by ordinary object-extension, or by a method in R which performs a self-inflicted extension.

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

The list of "reserved" methods in an object-type is crucial to enforce the consistency of the type assignment system. Consider for instance an object containing two methods, `add_n_1`, and `add_n_2`, say, 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 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.

In order to model specialization of inherited methods, we use the notion of *matching* or type extension, originally introduced by Bruce [11], and later applied to the Object Calculus [1] and to $\lambda\mathcal{O}bj$ [3]. At the price of a little more mathematical overhead, we could have used also the implicit higher-order polymorphism of [16].

Object Subsumption.

As it is well-known, see *e.g.* [1, 17], the introduction of a subsumption relation over object-types makes the type system unsound. In particular, width-subtyping clashes with object extension, and depth-subtyping clashes with object override. In fact on `pro`-types no subtyping is possible. In order to accommodate subtyping, we add another kind of object-type, *viz.* $\texttt{obj}\,t.\langle\!\langle R \triangleleft R'\rangle\!\rangle$, which behaves like $\texttt{pro}\,t.\langle\!\langle R \triangleleft R'\rangle\!\rangle$ except that it can be assigned to objects that can be extended only by methods in R' [1]. On `obj`-types a (covariant) width-subtyping is permitted.

[1] The `pro` and `obj` terminology is borrowed from Fisher and Mitchell [18, 19].

This paper is organized as follows. In Section 2 we introduce the calculus $\lambda\mathcal{O}bj+$; we present both a "Small-Step" reduction relation and an input/output, *viz* "Big-Step", operational semantics. Some intuitive examples are given, in order to illustrate the idea of a self-inflicted object extension. In Section 3 we introduce the type system for $\lambda\mathcal{O}bj+$. The intended meaning of the most interesting rules is discussed in detail. In Section 4 we show how our type system is compatible with a width-subtyping relation. A collection of example are presented in Section 2.2. In Section 6 we state our soundness result, namely that every closed and well-typed expression will not produce wrong results. In Section 7, we outline a possible application of our type system to the Object Calculus, in the style of [20]. Section 8 discuss related and future work. The complete set of type assignment rules appears in the Appendix.

Acknowledgment

The authors are grateful to Martin Abadi, Mariangiola Dezani-Ciancaglini and the anonymous referees for their useful comments on this work.

2 The Lambda Calculus of Objects

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

e	$::=$	$c \mid x \mid \lambda x.e \mid e_1\, e_2 \mid$	(λ-calculus)
		$\langle\,\rangle \mid e \Leftarrow \mathtt{m} \mid \langle e_1 \leftarrow\!\!\!\prec \mathtt{m} = e_2\rangle \mid$	(object terms)
		$Sel(e_1, \mathtt{m}, e_2)$,	(aux. operation)

where c is a meta-variable ranging over a set of constants, x is a meta-variable ranging over a set of variables and $\mathtt{m}$ is a meta-variable ranging over a set of methods names. As usual, terms that differ only in the names of bound variables are identified. The intended meaning of the object terms is the following: $\langle\,\rangle$ stands for the empty object; $e \Leftarrow \mathtt{m}$ stands for the result of sending the message $\mathtt{m}$ to the object e; $\langle e_1 \leftarrow\!\!\!\prec \mathtt{m} = e_2\rangle$ stands for extending/overriding the object e_1 with a method $\mathtt{m}$ whose body is e_2. The auxiliary operation $Sel(e_1, \mathtt{m}, e_2)$ searches the body of the $\mathtt{m}$ method within the object e_1, being e_1 a prototype of e_2. This function is peculiar to the operational semantics and, in practice, could be made not to be available to the programmer. This operation Sel, introduced in [21, 2, 7], provides a more direct *dynamic method lookup* than the *bookkeeping* reductions used in the original paper [16].

As we said in the introduction, the main difference between the syntax of $\lambda\mathcal{O}bj+$ and that of $\lambda\mathcal{O}bj$ ([16]) lies in the use of a single operator $\leftarrow\!\!\!\prec$, for building an object from an existing prototype. If the object e_1 contains $\mathtt{m}$, then $\leftarrow\!\!\!\prec$ denotes an object override, otherwise $\leftarrow\!\!\!\prec$ denotes an object extension.

2.1 Operational Semantics

In this section, we define the evaluation of the terms of $\lambda\mathcal{O}bj+$. We present two operational semantics. The first is given in terms of reduction rules (Small-Step), while the second is an input/output relation (Big-Step).

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

Figure 1: Reduction Semantics (Small-Step)

Reduction Semantics.

The core of the Small-Step reduction is given by the reduction rules of Figure 2.1. The evaluation relation $\stackrel{ev}{\twoheadrightarrow}$ is then taken to be the symmetric, reflexive, transitive and contextual closure of $\stackrel{ev}{\to}$.

In addition to the standard $(Beta)$ rule for lambda calculus, the main operation on objects is method invocation, whose reduction is defined by the $(Select)$ rule. Sending a message $\mathtt{m}$ to an object e containing a method $\mathtt{m}$ reduces to $Sel(e, \mathtt{m}, e)$, where the arguments of Sel have the following intuitive meanings (in reverse order):

- (3^{rd}-arg) is the receiver (or recipient) of the message;
- (2^{nd}-arg) is the message we want to send to the receiver of the message;
- (1^{st}-arg) is (or reduces to) a proper sub-object of the receiver of the message.

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

Using standard techniques in term rewriting systems we have:

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

Big-Step Operational Semantics.

We define an operational semantics by means of a natural proof deduction system à la Plotkin [22], which maps every input term (*i.e.* a closed term) to its output (*i.e.* a value). This semantics enforces a "lazy" evaluation strategy over terms. It is worth noticing that this semantics is *deterministic* and hence it specifies the behavior of a "possible interpreter" for $\lambda\mathcal{O}bj+$.

The set of values is defined as follows:

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

The deduction rules are presented in Figure 2.1. The Big-Step operational semantics is sound with respect to the Small-Step reduction in the following sense:

$$\frac{}{v \Downarrow v} \quad (Big{-}Value)$$

$$\frac{e_1 \Downarrow \lambda x.e'_1 \qquad [e_2/x]e'_1 \Downarrow v}{e_1\, e_2 \Downarrow v} \quad (Big{-}Beta)$$

$$\frac{e \Downarrow obj \qquad Sel(obj, \mathtt{m}, obj) \Downarrow v}{e \Leftarrow \mathtt{m} \Downarrow v} \quad (Big{-}Select)$$

$$\frac{e_2\, obj \Downarrow v}{Sel(\langle e_1 \leftarrow\!\!\!\!\prec \mathtt{m} = e_2\rangle, \mathtt{m}, obj) \Downarrow v} \quad (Big{-}Success)$$

$$\frac{e_1 \Downarrow obj' \qquad Sel(obj', \mathtt{m}, obj) \Downarrow v \qquad \mathtt{m} \neq \mathtt{n}}{Sel(\langle e_1 \leftarrow\!\!\!\!\prec \mathtt{n} = e_2\rangle, \mathtt{m}, obj) \Downarrow v} \quad (Big{-}Next)$$

Figure 2: Input/Output Semantics (Big-Step)

Proposition 2.2 (Soundness of $\Downarrow$) *If $e \Downarrow v$, then $e \overset{ev}{\twoheadrightarrow} v$.*
Proof: *By induction on the structure of the derivation of $e \Downarrow v$.*

We conjecture also the following:

Conjecture 2.3 (Completeness of $\Downarrow$) *If $e \overset{ev}{\twoheadrightarrow} v$, then there exists v' such that $e \Downarrow v'$.*

2.2 A Collection of Examples

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

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

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

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

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

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

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

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

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

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

$$\begin{aligned} &\mathtt{inner\_ext} \Leftarrow \mathtt{add\_m\_n} \overset{ev}{\twoheadrightarrow} \\ &\langle \mathtt{inner\_ext} \leftarrow\!\!\!\!\prec \mathtt{m} = \lambda \mathtt{s}.\langle \mathtt{s} \leftarrow\!\!\!\!\prec \mathtt{n} = \lambda \mathtt{s}'.1\rangle\rangle, \end{aligned}$$

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

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

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

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

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

$$\begin{aligned} &\mathtt{fly\_ext} \Leftarrow \mathtt{get\_f} \overset{ev}{\to} Sel(\mathtt{fly\_ext}, \mathtt{get\_f}, \mathtt{fly\_ext}) \\ &\overset{ev}{\to} \;\; (\lambda \mathtt{self}.(\mathtt{self} \Leftarrow \mathtt{f})\langle \mathtt{self} \leftarrow\!\!\!\!\prec \mathtt{n} = \lambda \mathtt{s}.1\rangle)\, \mathtt{fly\_ext} \\ &\overset{ev}{\to} \;\; (\mathtt{fly\_ext} \Leftarrow \mathtt{f})\langle \mathtt{fly\_ext} \leftarrow\!\!\!\!\prec \mathtt{n} = \lambda \mathtt{s}.1\rangle \\ &\overset{ev}{\to} \;\; Sel(\mathtt{fly\_ext}, \mathtt{f}, \mathtt{fly\_ext})\, \langle \mathtt{fly\_ext} \leftarrow\!\!\!\!\prec \mathtt{n} = \lambda \mathtt{s}.1\rangle \\ &\overset{ev}{\twoheadrightarrow} \;\; (\lambda \mathtt{self}.\lambda \mathtt{p}.\mathtt{p} \Leftarrow \mathtt{n})\, \mathtt{fly\_ext}\, \langle \mathtt{fly\_ext} \leftarrow\!\!\!\!\prec \mathtt{n} = \lambda \mathtt{s}.1\rangle \\ &\overset{ev}{\twoheadrightarrow} \;\; \langle \mathtt{fly\_ext} \leftarrow\!\!\!\!\prec \mathtt{n} = \lambda \mathtt{s}.1\rangle \Leftarrow \mathtt{n} \\ &\overset{ev}{\twoheadrightarrow} \;\; 1, \end{aligned}$$

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

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

3 The Type System

In this section, we will introduce the syntax of types, together with the most interesting type rules. In the sake of simplicity, we prefer to first present of the type system without the rules related with object subsumption (they 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:

$$\begin{array}{lcl} \sigma, \tau & ::= & \iota \mid t \mid \sigma \to \tau \mid \mathtt{pro}\, t.\langle\!\langle R \triangleleft R' \rangle\!\rangle \mid \sigma \twoheadleftarrow \mathtt{m} \\ R, R' & ::= & \varepsilon \mid R, \mathtt{m} : \sigma \\ \kappa & ::= & T \end{array}$$

where σ and τ are meta-variables ranging over types, ι is a meta-variable over constant types, R, and R' are meta-variables ranging over rows, and κ is a metavariable ranging over the unique kind of types, *i.e.* T. A *row* is an *unordered* set of pairs *(method label, method type)*. Object-types in the form $\texttt{pro}\, t.\langle\!\langle R \triangleleft R' \rangle\!\rangle$ are called pro-types, here pro is a binder for the type-variable t. As in [16], we may consider object-types as a form of recursively-defined types. In this paper we will freely use α-conversion of type-variables bound by pro.

The intended meaning of an object-type:

$$\texttt{pro}\, t.\langle\!\langle \texttt{m}_1 : \sigma_1 \ldots \texttt{m}_h : \sigma_h \triangleleft \texttt{m}_{h+1} : \sigma_{h+1} \ldots \texttt{m}_k : \sigma_k \rangle\!\rangle,$$

is the following:

- $\texttt{m}_1, \ldots, \texttt{m}_h$ are the methods that can be invoked; we say that these methods belong to the *interface* part of the object-type. Therefore, the object-type $\texttt{pro}\, t.\langle\!\langle \texttt{m}_1 : \sigma_1 \ldots \texttt{m}_h : \sigma_h \triangleleft \rangle\!\rangle$ is the counterpart of the object-type $\texttt{class}\, t.\langle\!\langle \texttt{m}_1 : \sigma_1 \ldots \texttt{m}_h : \sigma_h \rangle\!\rangle$ of [16].

- $\texttt{m}_{h+1}, \ldots, \texttt{m}_k$ are methods that cannot be invoked; they are *reserved*, *i.e.* they belong to the *reservation* part of the object-type. 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 $\texttt{pro}\, t.\langle\!\langle R \triangleleft R', \texttt{m} : \sigma \rangle\!\rangle$. As we remarked in the introduction, this reservation mechanism is crucial to guarantee the consistency of the type system.

The operator $\twoheadleftarrow$ is used to add new methods to an object-type; essentially it is the "type counterpart" of the operator $\leftarrow\!\!\!\prec$.

3.2 Contexts and Judgments

The contexts have the following shape:

$$\boxed{\Gamma ::= \varepsilon \mid \Gamma, x : \tau \mid \Gamma, t <\!\!\# \tau}$$

Our type assignment system uses judgments of the following shapes:

$$\boxed{\Gamma \vdash ok, \;\; \Gamma \vdash \sigma : T, \;\; \Gamma \vdash e : \sigma, \;\; \Gamma \vdash \sigma <\!\!\# \tau, \;\; \Gamma \vdash \sigma \stackrel{type}{\longrightarrow} \tau}$$

The intended meaning of the first three judgments is the natural one: well-formed contexts and types, and assignment of σ to an expression e.

The intended meaning of the judgment $\Gamma \vdash \sigma <\!\!\# \tau$ is that σ is the type of a possible extension of an object having type τ. As in [11, 3], this judgment formalizes the notion of *method-specialization* (or *protocol-extension*), *i.e.* the capability to "inherit" the type of the methods of the prototype.

The judgment $\Gamma \vdash \sigma \stackrel{type}{\longrightarrow} \tau$ expresses a limited form of type-conversion which amounts to simplify occurrences of $\twoheadleftarrow$. A formal system, instead of simple rewriting rules, is needed since bounded type-variables can occur into the reduction.

The type rules for well-formed contexts are standard and need no comment.

3.3 Well-formed Types Rules

In the following we will indicate with $\mathcal{M}(\texttt{m}_1 : \sigma_1 \ldots \texttt{m}_h : \sigma_h)$ the set $\{\texttt{m}_1 \ldots \texttt{m}_h\}$.

The $(Type\!-\!Pro_R)$ rule

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

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

The $(Type\!-\!Pro_L)$ rule

$$\frac{\Gamma \vdash \texttt{pro}\, t.\langle\!\langle \triangleleft R, R' \rangle\!\rangle : T}{\Gamma \vdash \texttt{pro}\, t.\langle\!\langle R \triangleleft R' \rangle\!\rangle : T}$$

asserts that methods (with related types) can be moved from the reservation to the interface part of an object-type without affecting the well-formedness of the object-type itself.

The $(Type\!-\!Extend)$ rule

$$\frac{\Gamma \vdash \tau <\!\!\# \,\texttt{pro}\, t.\langle\!\langle \triangleleft R, \texttt{m} : \sigma \rangle\!\rangle}{\Gamma \vdash \tau \twoheadleftarrow \texttt{m} : T}$$

asserts that, in order to apply $\twoheadleftarrow \texttt{m}$ to an object-type τ, the type τ needs to contain the method m.

3.4 Matching Type Rules

The most interesting rules for method-specialization are $(Match\!-\!Inherit)$, $(Match\!-\!Book\!-\!Pro)$, $(Match\!-\!Red_L)$, and $(Match\!-\!Red_R)$.

The $(Match\!-\!Inherit)$ rule

$$\frac{\Gamma \vdash \tau \twoheadleftarrow \texttt{m} : T}{\Gamma \vdash \tau \twoheadleftarrow \texttt{m} <\!\!\# \tau}$$

asserts that a type τ incremented with a method m is a specialization of the original type τ. This rule captures the essence of protocol-extension. Note that τ can be a type-variable declared in the context Γ.

The $(Match\!-\!Book\!-\!Pro)$ rule

$$\frac{\Gamma \vdash \texttt{pro}\, t.\langle\!\langle R \triangleleft R' \rangle\!\rangle : T \qquad \Gamma \vdash \texttt{pro}\, t.\langle\!\langle R \triangleleft R', \texttt{m} : \sigma \rangle\!\rangle : T}{\Gamma \vdash \texttt{pro}\, t.\langle\!\langle R \triangleleft R', \texttt{m} : \sigma \rangle\!\rangle <\!\!\# \,\texttt{pro}\, t.\langle\!\langle R \triangleleft R' \rangle\!\rangle}$$

asserts that an object-type with less reserved methods can be specialized to an object-type with more reserved methods.

The $(Match\!-\!Red_L)$, and $(Match\!-\!Red_R)$ rules are simple: two equivalent types can be considered as being one the specialization of the other.

The remaining rules are the usual rules for reflexivity, transitivity, and contextual closure with respect to $\twoheadleftarrow$.

3.5 Type Reduction Rules

The type-reduction rules allow to transform an object-type in an equivalent, but syntactically simpler, type. In this subsection, we discuss the rules $(Red\!-\!Over)$, and $(Red\!-\!Ext)$.

The $(Red\!-\!Over)$ rule

$$\frac{\Gamma \vdash \tau <\!\!\# \,\texttt{pro}\, t.\langle\!\langle R, \texttt{m} : \sigma \triangleleft R' \rangle\!\rangle}{\Gamma \vdash \tau \twoheadleftarrow \texttt{m} \stackrel{type}{\longrightarrow} \tau}$$

asserts that, if in a type τ the method m is already present in the interface part, then the type $\tau \twoheadleftarrow \mathtt{m}$ can be simplified to τ. The type reduction judgment, instead of a single set of rewriting rules, is necessary precisely because, in this rule, the type τ can be also a type variable t.

The $(Red{-}Ext)$ rule

$$\frac{\Gamma \vdash \mathtt{pro}\, t.\langle\langle R \lhd R', \mathtt{m} : \sigma \rangle\rangle : T}{\Gamma \vdash \mathtt{pro}\, t.\langle\langle R \lhd R', \mathtt{m} : \sigma \rangle\rangle \twoheadleftarrow \mathtt{m} \xrightarrow{type} \mathtt{pro}\, t.\langle\langle R, \mathtt{m} : \sigma \lhd R' \rangle\rangle}$$

asserts that an object-type τ, where the method m is reserved, incremented with the method m, reduces to an object-type where the method m is *shifted* to the interface part.

The remaining rules enforce the contextual closure of $\xrightarrow{type}$.

3.6 Type Rules for Terms

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

The $(Pre{-}Extend)$ rule

$$\frac{\Gamma \vdash e : \mathtt{pro}\, t.\langle\langle R \lhd R' \rangle\rangle \qquad \Gamma \vdash \mathtt{pro}\, t.\langle\langle R \lhd R', \mathtt{m} : \sigma \rangle\rangle : T}{\Gamma \vdash e : \mathtt{pro}\, t.\langle\langle R \lhd R', \mathtt{m} : \sigma \rangle\rangle}$$

asserts that an object having type $\mathtt{pro}\, t.\langle\langle R \lhd R' \rangle\rangle$ can be considered also as an object having type $\mathtt{pro}\, t.\langle\langle R \lhd R', \mathtt{m} : \sigma \rangle\rangle$. This rule has to be used in conjunction with the rule $(Object)$; it ensures that we can dynamically add fresh methods. This rule cannot be applied when e is the variable `self`. In fact, as explained in Remark 3.1, the type of `self` can only be a type variable. This fact is crucial for the soundness of the type system.

The $(Object)$ rule

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \tau \\ \Gamma \vdash \tau <\!\# \,\mathtt{pro}\, t.\langle\langle R \lhd R', \mathtt{m} : \sigma \rangle\rangle \\ \Gamma, t <\!\# \,\tau \twoheadleftarrow \mathtt{m} \vdash e_2 : t \to \sigma\end{array}}{\Gamma \vdash \langle e_1 \leftarrow\!\!\!\prec \mathtt{m} = e_2 \rangle : \tau \twoheadleftarrow \mathtt{m}}$$

can be applied in the following cases:

1. when the object e_1 has type *e.g.* $\mathtt{pro}\, t.\langle\langle R, \mathtt{m} : \sigma \lhd R' \rangle\rangle$. In this case the method m is *already* present in the object e_1, and the body of m is *overridden* with a new method body;

2. when the object e_1 has type $\mathtt{pro}\, t.\langle\langle R \lhd R', \mathtt{m} : \sigma \rangle\rangle$ (or, by a previous application of the *(Pre-Extend)* rule, $\mathtt{pro}\, t.\langle\langle R \lhd R' \rangle\rangle$). In this case the object e_1 is *extended* with the (fresh) method m;

3. when τ is a type variable t. In this case e_1 can be the variable `self` and there is a *self-inflicted extension*.

The bound for t is the same as the final type for the object $\langle e_1 \leftarrow\!\!\!\prec \mathtt{m} = e_2 \rangle$; this guarantees that the method m specializes its type for *every future extension* of $\langle e_1 \leftarrow\!\!\!\prec \mathtt{m} = e_2 \rangle$.

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

The $(Send)$ rule

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash \tau <\!\# \,\mathtt{pro}\, t.\langle\langle R, \mathtt{m} : \sigma \lhd R' \rangle\rangle}{\Gamma \vdash e \Leftarrow \mathtt{m} : \sigma[\tau/t]}$$

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

The $(Select)$ is the following:

$$\frac{\begin{array}{ll}\Gamma \vdash e_1 : \tau & \Gamma \vdash e_2 : \rho \\ \Gamma \vdash \tau <\!\# \,\mathtt{pro}\, t.\langle\langle R, \mathtt{m} : \sigma \lhd R' \rangle\rangle & \Gamma \vdash \rho <\!\# \,\tau\end{array}}{\Gamma \vdash Sel(e_1, \mathtt{m}, e_2) : \sigma[\rho/t]}$$

The leftmost two conditions in this rule ensure that method m is in the interface part of the object e_1, while the other two conditions ensure that e_1 is a prototype of e_2.

The (Red_L) and (Red_R) states a form of type conversion. if an object can be typed with σ, then it can be typed also with a type τ equivalent to σ.

4 Adding Object Subsumption

In this section we propose an extension of the type assignment system for $\lambda Obj+$ to accommodate width-subtyping.

It is well known that adding a subsumption rule over terms, that is the rule $(Subsume)$

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

clearly increases the set of expressions which are typable in the type system. Unfortunately, this rule is *unsound* in the presence of object extension; in fact, we could (by subsumption) first hide a method in an object, and then add it again with a type incompatible with the previous one, see [1, 17].

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

$$\mathtt{obj}\, t.\langle\langle R \lhd R' \rangle\rangle,$$

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 $\mathtt{obj}\, t.\langle\langle R \lhd R' \rangle\rangle$, it follows that the type $\mathtt{obj}\, t.\langle\langle R \lhd R' \rangle\rangle$ permits extensions of an object only with respect to the method names listed in the reservation part R'. This approach to subsumption is reminiscent of the one in [18, 20].

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

Therefore, the new syntax of types and kinds is :

$$\begin{array}{lcl}\sigma, \tau & ::= & \ldots \mid \mathtt{obj}\, t.\langle\langle R \lhd R' \rangle\rangle \\ \kappa & ::= & T \mid Rgd.\end{array}$$

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

$$\Gamma \vdash \sigma : Rgd.$$

The intuitive typing rules related with this judgment are presented in the Appendix. The (*Subsume*) rule is valid only when the type in the conclusion is rigid. It is important to point out that, by 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 so far in literature ([11, 10, 3]). Hence we can suggestively say that in our type assignment system *"matching is a relation on types compatible with a limited subsumption rule"*.

4.1 Extra Rules for Subsumption

Extra rules for the obj-types are necessary. Most of these rules are simply a rephrasing of the rules presented so far, replacing the binder pro by the binder obj.

The most important new rules are (*Type−Obj*), and (*Promote*).

The (*Type−Obj*) rule

$$\frac{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R \triangleleft R' \rangle\!\rangle : T \qquad t \text{ covariant in } R, R'}{\Gamma \vdash \mathtt{obj}\, t.\langle\!\langle R \triangleleft R' \rangle\!\rangle : T}$$

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 occurs only covariantly in the methods of R, and R'. Hence, *binary methods* are lost. This is, unfortunately, a common price to pay in order to have a fully static type system with subtyping (see [8, 13, 14]).

The (*Promote*) rule

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

"promotes" a fully-specializable pro-type into a limitedly specializable obj-type. When the methods in R' are added to the object assigned to an obj-type, the object in question becomes *fixed*, as in the Object Calculus.

5 A Collection of Examples

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

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

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

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

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

A possible derivation for self_ext is presented in Figure 5.

Example 5.1 This example shows how class declaration can be simulated in the lambda calculus of object and how using the self-inflicted extension we can factorize in a single declaration the definition of a hierarchy of classes. Let the method add_set_col *be defined as in Example 1.2, and let us consider the simple class definition of* PointClass*:*

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

The object PointClass *can be used to create instances of both points and colored points, by using the expressions:* $\mathtt{PointClass} \Leftarrow \mathtt{new}$ *and* $(\mathtt{PointClass} \Leftarrow \mathtt{new}) \Leftarrow \mathtt{add\_set\_col}$.

Example 5.2 (Subsumption 1) This example shows how subsumption can interact with object extension. Let:

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

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

By the type assignment rules we have:

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

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

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

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

Notice that the object:

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

would not be typable without the subsumption rules.

Example 5.3 (Subsumption 2) This example shows how a self-inflicted extension can interact with object subsumption. Let:

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

By the type assignment rules we have:

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

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

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

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

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

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

$$\varepsilon \vdash \mathtt{point} : \mathtt{pro}\, t.R \quad \textit{and} \quad \varepsilon \vdash \mathtt{col\_point} : \mathtt{pro}\, t.R',$$

where R *and* R' *are, respectively*

$$\begin{array}{l} \langle\!\langle \mathtt{x} : \mathtt{int}, \quad \mathtt{equal} : t \to \mathtt{bool}, \mathtt{add\_set\_col} : t \twoheadleftarrow \mathtt{col} \triangleleft \\ \qquad\qquad \mathtt{col} : \mathtt{bool} \rangle\!\rangle \\ \langle\!\langle \mathtt{x} : \mathtt{int}, \mathtt{equal} : t \to \mathtt{bool}, \mathtt{add\_set\_col} : t, \mathtt{col} : \mathtt{bool} \triangleleft \rangle\!\rangle. \end{array}$$

The following judgments are derivable:

$$\varepsilon \vdash \mathtt{col\_point} \Leftarrow \mathtt{equal} : \mathtt{pro}\, t.R' \to \mathtt{bool}$$

$$\varepsilon \vdash \mathtt{point} \Leftarrow \mathtt{add\_set\_col} : \mathtt{pro}\, t.R \twoheadleftarrow \mathtt{col} \xrightarrow{type} \mathtt{pro}\, t.R'$$

$$\varepsilon \vdash \mathtt{col\_point} \Leftarrow \mathtt{equal}(\mathtt{point} \Leftarrow \mathtt{add\_set\_col}) : \mathtt{bool}.$$

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{\Gamma_2 \vdash \mathtt{s} : v \qquad \Gamma_2 \vdash v <\# \mathtt{pro}\, t.\langle\!\langle \triangleleft \mathtt{n} : \mathtt{int} \rangle\!\rangle \qquad \Gamma_3, \mathtt{s}' : w \vdash 1 : \mathtt{int}}{\Gamma_2 \vdash \langle \mathtt{s} \longleftarrow \mathtt{n} = \lambda \mathtt{s}'.1 \rangle : v \twoheadleftarrow \mathtt{n}}\ (Obj)
}{\Gamma_1 \vdash \lambda \mathtt{s}.\langle \mathtt{s} \longleftarrow \mathtt{n} = \lambda \mathtt{s}'.1 \rangle : (v \to (v \twoheadleftarrow \mathtt{n}))}\ (Abs)
\qquad \varepsilon \vdash \langle\,\rangle : \tau^{*} \qquad \varepsilon \vdash \tau <\# \tau
}{\varepsilon \vdash \langle \mathtt{add\_n} = \lambda \mathtt{s}.\langle \mathtt{s} \longleftarrow \mathtt{n} = \lambda \mathtt{s}'.1 \rangle \rangle : \tau \twoheadleftarrow \mathtt{add\_n}}\ (Obj)
\qquad \varepsilon \vdash \tau \twoheadleftarrow \mathtt{add\_n} \xrightarrow{type} \mathtt{pro}\, t.\langle\!\langle \mathtt{add\_n} : t \twoheadleftarrow \mathtt{n} \triangleleft \mathtt{n} : \mathtt{int} \rangle\!\rangle
}{\varepsilon \vdash \langle \mathtt{add\_n} = \lambda \mathtt{s}.\langle \mathtt{s} \longleftarrow \mathtt{n} = \lambda \mathtt{s}'.1 \rangle \rangle : \mathtt{pro}\, t.\langle\!\langle \mathtt{add\_n} : t \twoheadleftarrow \mathtt{n} \triangleleft \mathtt{n} : \mathtt{int} \rangle\!\rangle}\ (Red)
$$

where

$$\tau \equiv \mathtt{pro}\, t.\langle\!\langle \triangleleft\, \mathtt{add\_n} : t \twoheadleftarrow \mathtt{n}, \mathtt{n} : \mathtt{int} \rangle\!\rangle, \quad \Gamma_1 \equiv v <\# \tau \twoheadleftarrow \mathtt{add\_n}, \quad \Gamma_2 \equiv \Gamma_1, \mathtt{s} : v, \quad \text{and} \quad \Gamma_3 \equiv \Gamma_2, w <\# v \twoheadleftarrow \mathtt{n},$$

and the judgment $(*)$ is derivable by two applications of the rule $(Pre{-}Extend)$.

Figure 3: A derivation for `self_ext`

6 Soundness of the Type System

In this section we prove the crucial property of our type system, *i.e.* Theorem 6.11, the subject reduction theorem. First we need a series of technical lemmata which can be proved by complex albeit standard inductive arguments. As a corollary of the Theorem 6.11, we shall derive the fundamental result of the paper: *i.e.* the type soundness of our typing discipline.

Since every reduction step eliminates one occurrence of the operator $\twoheadleftarrow$, and since there are no critical pairs, it follows that:

Proposition 6.1 *The type-reduction $\Gamma \vdash \sigma \xrightarrow{type} \tau$ is Church-Rosser and Strongly Normalizing.*

In the following we will first consider proof the subject reduction theorem for the *plain* type assignment system not containing subtyping.

In the statement of the following properties and theorems, we will denote by A either ok, or $\sigma : T$, or $\sigma <\# \tau$, or $\sigma \xrightarrow{type} \tau$, or $e : \sigma$.

Lemma 6.2 *(i) If $\Gamma \vdash A$, then $\Gamma \vdash ok$;*

(ii) if $\Gamma, t <\# \tau, \Gamma' \vdash ok$, then $\Gamma \vdash \tau : T$.

Proof: Both points can be easily proved by structural induction on the derivation of judgments. □

In the sequel, we make use of the following abbreviations:

- $\Gamma \vdash \mathtt{m} \in \mathcal{M}(\tau)$ stands for $\Gamma \vdash \tau <\# \mathtt{pro}\, t.\langle\!\langle \triangleleft R, \mathtt{m} : \sigma \rangle\!\rangle$, for some R, σ.
- $\Gamma \vdash \sigma \overset{type}{=} \sigma'$ stands for $\Gamma \vdash \sigma \overset{type}{\twoheadrightarrow} \sigma''$, and $\Gamma \vdash \sigma' \overset{type}{\twoheadrightarrow} \sigma''$, for some σ''. Here $\overset{type}{\twoheadrightarrow}$ is the transitive closure of $\xrightarrow{type}$.

Lemma 6.3 *If $\Gamma \vdash \mathtt{m} \in \mathcal{M}(\tau \twoheadleftarrow \mathtt{n})$, then $\Gamma \vdash \mathtt{m} \in \mathcal{M}(\tau)$.*

Proof: By inspection on the derivation of

$$\Gamma \vdash \tau \twoheadleftarrow \mathtt{n} <\# \mathtt{pro}\, t.\langle\!\langle \triangleleft R, \mathtt{m} : \sigma \rangle\!\rangle.$$ □

Lemma 6.4 *If $\Gamma \vdash \mathtt{m} \in \mathcal{M}(\tau)$ and $\Gamma \vdash \tau \xrightarrow{type} \sigma$, then $\Gamma \vdash \mathtt{m} \in \mathcal{M}(\sigma)$.*

Proof: By structural induction on the derivation of $\Gamma \vdash \tau \xrightarrow{type} \sigma$. The induction hypothesis needs to be applied only in the case of rule $(Red{-}Inherit)$. When considering the rules $(Red{-}Over)$, $(Red{-}Ext)$, and $(Red{-}Inherit)$ one needs to apply Lemma 6.3. □

Lemma 6.5 *If $\Gamma \vdash \sigma <\# \tau$, or $\Gamma \vdash \sigma \xrightarrow{type} \tau$, then $\Gamma \vdash \sigma : T$, and $\Gamma \vdash \tau : T$.*

Proof: By structural induction on the derivations of $\Gamma \vdash \sigma <\# \tau$, or $\Gamma \vdash \sigma \xrightarrow{type} \tau$. In order to deal with the rule $(Red{-}Inherit)$, one needs to apply Lemma 6.4. □

Lemma 6.6 *If $\Gamma, t <\# \tau, \Gamma' \vdash A$, and $\Gamma \vdash \sigma <\# \tau$, then $\Gamma, t <\# \sigma, \Gamma' \vdash A$.*

Proof: The proof is immediate. □

Lemma 6.7 (Substitution) *(i) If $\Gamma, t <\# \tau, \Gamma' \vdash A$, and $\Gamma \vdash \sigma <\# \tau$, then $\Gamma, \Gamma'[\sigma/t] \vdash A[\sigma/t]$.*

(ii) If $\Gamma, x : \sigma, \Gamma' \vdash e : \tau$, and $\Gamma \vdash e' : \sigma$, then $\Gamma, \Gamma' \vdash e[e'/x] : \tau$.

Proof: By structural induction on the derivation of judgment $\Gamma, t <\# \tau, \Gamma' \vdash A$ for point (i) or $\Gamma, x : \sigma, \Gamma' \vdash e : \tau$ for point (ii). □

Lemma 6.8 *(i) If $\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R \triangleleft R' \rangle\!\rangle \twoheadleftarrow \mathtt{n}_1 \ldots \twoheadleftarrow \mathtt{n}_k \overset{type}{\twoheadrightarrow} \mathtt{pro}\, t.\langle\!\langle R'' \triangleleft R''' \rangle\!\rangle \twoheadleftarrow \mathtt{n}'_1 \ldots \twoheadleftarrow \mathtt{n}'_{k'}$, then $\mathtt{m} : \sigma' \in R'', R'''$ if and only if there exists σ such that: $\mathtt{m} : \sigma \in R, R'$ and $\Gamma, t <\# \mathtt{pro}\, t.\langle\!\langle R \triangleleft R' \rangle\!\rangle \twoheadleftarrow \mathtt{n}_1 \ldots \twoheadleftarrow \mathtt{n}_k \vdash \sigma \overset{type}{=} \sigma'$;*

(ii) If $\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R \triangleleft R' \rangle\!\rangle \twoheadleftarrow \mathtt{n}_1 \ldots \twoheadleftarrow \mathtt{n}_k \overset{type}{=} \mathtt{pro}\, t.\langle\!\langle R'' \triangleleft R''' \rangle\!\rangle \twoheadleftarrow \mathtt{n}'_1 \ldots \twoheadleftarrow \mathtt{n}'_{k'}$ and $\mathtt{m} : \sigma' \in R'', R'''$ then there exists σ such that: $\mathtt{m} : \sigma \in R, R'$ and $\Gamma, t <\# \mathtt{pro}\, t.\langle\!\langle R \triangleleft R' \rangle\!\rangle \twoheadleftarrow \mathtt{n}_1 \ldots \twoheadleftarrow \mathtt{n}_k \vdash \sigma \overset{type}{=} \sigma'$.

Proof: Both points can be proved by structural induction on the derivation. To deal with the transitive closure of the matching relation, it is necessary to use Lemma 6.6. □

Lemma 6.9 *If* $\Gamma \vdash \tau <\!\#\, \mathtt{pro}\, t.\langle\!\langle \triangleleft R, \mathtt{m} : \sigma \rangle\!\rangle$*, and* $\Gamma \vdash \tau <\!\#\, \mathtt{pro}\, t.\langle\!\langle \triangleleft R', \mathtt{m} : \sigma' \rangle\!\rangle$*, then* $\Gamma, t <\!\#\, \tau \vdash \sigma \overset{type}{=} \sigma'$*.*

Proof: The lemma follows immediately from Lemmas 6.8, and 6.7 (the second lemma is needed when τ is in the form $t \twoheadleftarrow \mathtt{n}_1 \ldots \twoheadleftarrow \mathtt{n}_k$). □

Lemma 6.10 *If* $\Gamma \vdash e : \tau$*, then* $\Gamma \vdash \tau : T$*.*

Proof: By structural induction on the derivation of $\Gamma \vdash e : \tau$, using Lemma 6.7. □

Finally, using the above lemmata we can establish:

Theorem 6.11 (Subject Reduction) *If* $\Gamma \vdash e : \sigma$*, and* $e \overset{ev}{\to} e'$*, then* $\Gamma \vdash e' : \sigma$*.*

Proof: We proof that the type is preserved by each of the four reduction rules. In the case of the (*Beta*) rule the proof follows easily by the Substitution Lemma. In the case of the (*Select*) rule the proof is immediate. In the case of the (*Succ*) rule one need to consider the derivation of the judgment: $\Gamma \vdash Sel(\langle e_1 \leftarrow\!\!- \mathtt{m} = e_2\rangle, \mathtt{m}, e) : \sigma$. This judgment must be derived by an application of the (*Select*) rule (in case followed by several application of the (*Pre−Extend*) and (*Red*) rules). We have therefore:
$\Gamma \vdash Sel(\langle e_1 \leftarrow\!\!- \mathtt{m} = e_2\rangle, \mathtt{m}, e) : \sigma[\rho/t]$,
$\Gamma \vdash \langle e_1 \leftarrow\!\!- \mathtt{m} = e_2\rangle : \tau$, $\Gamma \vdash \tau <\!\#\, \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma \triangleleft R' \rangle\!\rangle$,
$\Gamma \vdash e : \rho$, and $\Gamma \vdash \rho <\!\#\, \tau$.
The judgment $\Gamma \vdash \langle e_1 \leftarrow\!\!- \mathtt{m} = e_2\rangle : \tau$ must be derived by an application of the (*Object*) rule (in case followed by several application of the (*Pre−Extend*) and (*Red*) rules). We have therefore: $\Gamma \vdash \langle e_1 \leftarrow\!\!- \mathtt{m} = e_2\rangle : \tau' \twoheadleftarrow \mathtt{m}$, $\Gamma \vdash e_1 : \tau'$, $\Gamma \vdash \tau' <\!\#\, \mathtt{pro}\, t.\langle\!\langle R'' \triangleleft R''', \mathtt{m} : \sigma'' \rangle\!\rangle$, and $\Gamma, t <\!\#\, \tau' \twoheadleftarrow \mathtt{m} \vdash e_2 : t \to \sigma''$. It follows that $\Gamma \vdash \tau' \twoheadleftarrow \mathtt{m} <\!\#\, \tau \twoheadleftarrow \mathtt{m}$ and from this, by transitivity of the matching relation, (*Red−Ext*) and (*Match−Red$_L$*) rules:
$\Gamma \vdash \tau' \twoheadleftarrow \mathtt{m} <\!\#\, \mathtt{pro}\, t.\langle\!\langle R'', \mathtt{m} : \sigma'' \triangleleft R''' \rangle\!\rangle$. By the Lemma 6.9 one can derived that $\Gamma, t <\!\#\, \tau \vdash \sigma \overset{type}{=} \sigma''$, It follow that: $\Gamma, t <\!\#\, \rho \vdash e_2 : t \to \sigma$ and by the Substitution Lemma we can easily conclude the proof. In the case of the (*Next*) rules one need to follows a similar pattern. □

The proof of subject reduction for the type assignment system with subtyping follows a similar pattern. All the auxiliary lemmata are valid in the type assignment system with subtyping, and the respective proofs can be straightforwardly extended. Moreover we have:

Lemma 6.12 *If* $\Gamma \vdash \tau <\!\#\, \mathtt{obj}\, t.\langle\!\langle \triangleleft R, \mathtt{m} : \sigma \rangle\!\rangle$*, and* $(\Gamma \vdash \tau <\!\#\, \mathtt{obj}\, t.\langle\!\langle \triangleleft R', \mathtt{m} : \sigma' \rangle\!\rangle \vee \Gamma \vdash \tau <\!\#\, \mathtt{pro}\, t.\langle\!\langle \triangleleft R', \mathtt{m} : \sigma' \rangle\!\rangle)$*, then* $\Gamma, t <\!\#\, \tau \vdash \sigma \overset{type}{=} \sigma'$*.*

In proving the Subject Reduction Theorem, one need to consider the case where, in the derivation of the typing judgment, there are applications of the (*Subsume*) rule. In this case it is sufficient to observe that after the application of a (*Subsume*) rule no further application of the (*Pre−Extend*) rule is possible.

We conclude this section by showing the type soundness theorem; this will guarantee that every closed and well typed expression will not produce the `message-not-found` runtime error. This error arises whenever we search a message `m` into an expression that does not reduce to an object which has the method `m` in its interface.

Definition 6.13 *Define the set of* `wrong` *terms as follows:*

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

By a direct inspection of the typing rules for terms, one can immediately see that that `wrong` cannot be typed. Hence, type soundness follows as a corollary of the subject reduction theorem, *i.e.*:

Corollary 6.14 (Type Soundness) *If* $\varepsilon \vdash e : \tau$*, then* $e \overset{ev}{\not\twoheadrightarrow} C[\mathtt{wrong}]$*, where* $C[\;]$ *is a generic context in* $\lambda\mathcal{O}bj+$ *i.e. a term with an "hole" inside it.*

7 Plug'n-Play in the Object Calculus

In [20] we are presented with a first-order extension of the Object Calculus of Abadi and Cardelli ([1]) that supports method extension in presence of object subsumption.

We conjecture that, with a little effort, the type system of $\lambda\mathcal{O}bj+$ could be adapted to the extended Object Calculus, of course taking into account that objects are set of pairs instead of lists in $\lambda\mathcal{O}bj+$. In order to adapt the type system of $\lambda\mathcal{O}bj+$ to the extended Object Calculus we need to:

- remove the (*Select*) rule;
- add the following (*Object*) rule in order to build an object from scratch:
  Let $\tau \equiv \mathtt{pro}\, t.\langle\!\langle R \triangleleft R' \rangle\!\rangle$:
  $$\frac{\Gamma, t <\!\#\, \tau, s : t \vdash e_i : \sigma_i \qquad \mathtt{m}_i : \sigma_i \in R \qquad \forall\, i \in I}{\Gamma \vdash \langle \mathtt{m}_i = \varsigma(s_i : t <\!\#\, \tau) e_i \rangle^{i \in I} : \mathtt{pro}\, t.\langle\!\langle R \triangleleft R' \rangle\!\rangle}$$
- modify the (*Object*) rule as follows:
  $$\frac{\begin{array}{l}\Gamma \vdash e_1 : \tau \\ \Gamma \vdash \tau <\!\#\, \mathtt{pro}\, t.\langle\!\langle R \triangleleft R', \mathtt{m} : \sigma \rangle\!\rangle \\ \Gamma, t <\!\#\, \tau \twoheadleftarrow \mathtt{m}, s : t \vdash e_2 : \sigma\end{array}}{\Gamma \vdash e_1.\mathtt{m} \leftarrow\!\!- \varsigma(s : t <\!\#\, \tau \twoheadleftarrow \mathtt{m}) e_2 : \tau \twoheadleftarrow \mathtt{m}}$$

8 Related and Future Works

Several calculi proposed in literature combine object extension with objects subsumption ([24, 18, 20, 4, 23]).

In [24], it is presented a calculus where it is possible to first subsume (forget) an object component, and then re-add it again with a type which may be incompatible with the forgotten one. In order to guarantee the soundness of the type system, *method dictionaries* are used inside objects, which link correctly method names and method bodies.

Approaches to subsumption similar to the one presented in this work can be found in [18, 20, 4, 23], In [20], an extension of the Object Calculus is presented. Roughly speaking we can say that `pro`-types and `obj`-types in this article correspond to "diamond-types" and "saturated-types" in [20]. Similar ideas can be found in [23], 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

Typed Syntax. $e ::= s \mid \langle \mathtt{m}_i = \varsigma(s_i : t <\!\#\, \tau) e_i \rangle^{i \in I} \mid e.\mathtt{m} \mid e_1.\mathtt{m} \leftarrow\!\!\prec \varsigma(s : t <\!\#\, \tau) e_2$

Operational Semantics. Let $e \triangleq \langle \mathtt{m}_i = \varsigma(s_i : t <\!\#\, \tau') e_i \rangle^{i \in I}$

$$(Sending) \quad e.\mathtt{m}_j \quad \overset{ev}{\to} \quad e_j\{e/s_j\} \qquad (j \in I)$$

$$(Object) \quad e.\mathtt{m}_j \leftarrow\!\!\prec \varsigma(s_j : t <\!\#\, \tau \twoheadleftarrow \mathtt{m}_j) e' \quad \overset{ev}{\to} \quad \langle \mathtt{m}_i = \varsigma(s_i : t <\!\#\, \tau \twoheadleftarrow \mathtt{m}_j) e_i, \mathtt{m}_j = \varsigma(s_j : t <\!\#\, \tau \twoheadleftarrow \mathtt{m}_j) e' \rangle^{i \in I - \{j\}} \qquad (j \in I)$$

Figure 4: Typed Syntax and Small-Step Semantics for the Extended Object Calculus

type is fixed in the declaration of the body of m. As a consequence the following expressions are not typable in that system:

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

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

Another type system for the Lambda Calculus of Objects is presented in [4]. This type system uses a refined notion of subtyping that allows to type also *binary methods*.

Interesting directions for our future work are the following. The type system of $\lambda\mathcal{O}bj+$ is not decidable. This problem can be solved by developing an "explicitly typed" version of $\lambda\mathcal{O}bj+$. In view of practical applications, it is also crucial to investigate the introduction of imperative features in our calculus. Finally we would like to find an equational theory dealing with the objects of $\lambda\mathcal{O}bj+$, and to study possible representation of our typing discipline in the framework theory of $F^{\omega}_{<:}$ (see [9]).

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects.* Springer-Verlag, 1996.

[2] G. Bellè. Some Remarks on Lambda Calculus of Objects. Technical report, Dipartimento di Matematica ed Informatica, Università di Udine, 1994.

[3] V. Bono and M. Bugliesi. Matching Constraints for the Lambda Calculus of Objects. In *Proc. of TLCA*, volume 1210 of *Lecture Notes in Computer Science*, pages 46–62. Springer-Verlag, 1997.

[4] V. Bono, M. Bugliesi, M. Dezani-Ciancaglini, and L. Liquori. Subtyping Constraint for Incomplete Objects. In *Proc. of TAPSOFT/CAAP*, volume 1214 of *Lecture Notes in Computer Science*, pages 465–477. Springer-Verlag, 1997.

[5] V. Bono, M. Bugliesi, and L. Liquori. A Lambda Calculus of Incomplete Objects. In *Proc. of MFCS*, volume 1113 of *Lecture Notes in Computer Science*, pages 218–229. Springer-Verlag, 1996.

[6] V. Bono and K. Fisher. An Imperative. First-Order Calculus with Object Extension. In *Proc. of ECOOP*, Lecture Notes in Computer Science. Springer-Verlag, 1998. To appear.

[7] V. Bono and L. Liquori. A Subtyping for the Fisher-Honsell-Mitchell Lambda Calculus of Objects. In *Proc. of CSL*, volume 933 of *Lecture Notes in Computer Science*, pages 16–30. Springer-Verlag, 1995.

[8] K. 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.

[9] K. Bruce, L Cardelli, and C. B. Pierce. Comparing Object Encoding. In *Proc. of TACS*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[10] K. Bruce, L. Petersen, and A. Fiech. Subtyping Is Not a Good "Match" for Object-Oriented Languages. In *Proc. of ECOOP*, volume 1241 of *Lecture Notes in Computer Science*, pages 104–127. Springer-Verlag, 1997.

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

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

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

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

[15] K. Fisher. *Type System for Object-Oriented Programming Languages.* PhD thesis, University of Stanford, August 1996.

[16] K. Fisher, F. Honsell, and J. C. Mitchell. A Lambda Calculus of Objects and Method Specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.

[17] K. Fisher and J. C. Michell. Notes on Typed Object-Oriented Programming. In *Proc. of TACS*, volume 789 of *Lecture Notes in Computer Science*, pages 844–885. Springer-Verlag, 1994.

[18] K. Fisher and J. C. Mitchell. A Delegation-based Object Calculus with Subtyping. In *Proc. of FCT*, volume 965 of *Lecture Notes in Computer Science*, pages 42–61. Springer-Verlag, 1995.

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

[20] L. Liquori. An Extended Theory of Primitive Objects: First Order System. In *Proc. of ECOOP*, volume 1241 of *Lecture Notes in Computer Science*, pages 146–169. Springer-Verlag, 1997.

[21] P. Paladin. Teoremi di Congruenza per Lambda-Calcoli Orientati agli Oggetti. Master's thesis, Dipartimento di Matematica ed Informatica, Università di Udine, 1993. In Italian.

[22] Gordon Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, Denmark, 1981.

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

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

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

A The Type Assignment Rules

Well-formed Contexts

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

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

$$\frac{\Gamma \vdash \sigma : T \qquad t \notin dom(\Gamma)}{\Gamma, t <\!\!\# \, \sigma \vdash ok} \quad (Cont{-}t)$$

Well-formed Types

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

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

$$\frac{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle \triangleleft R, R' \rangle\!\rangle : T}{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R \triangleleft R' \rangle\!\rangle : T} \quad (Type{-}Pro_L)$$

$$\frac{\Gamma \vdash \tau <\!\!\# \, \mathtt{pro}\, t.\langle\!\langle \triangleleft R, \mathtt{m} : \sigma \rangle\!\rangle}{\Gamma \vdash \tau \twoheadleftarrow \mathtt{m} : T} \quad (Type{-}Extend{-}Pro)$$

$$\frac{\Gamma, t <\!\!\# \, \sigma, \Gamma' \vdash ok}{\Gamma, t <\!\!\# \, \sigma, \Gamma' \vdash t : T} \quad (Type{-}Var)$$

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

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

Matching Rules

$$\frac{\Gamma \vdash \tau \twoheadleftarrow \mathtt{m} : T}{\Gamma \vdash \tau \twoheadleftarrow \mathtt{m} <\!\!\# \, \tau} \quad (Match{-}Inherit)$$

$(Match{-}Book{-}Pro)$

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

$$\frac{\Gamma \vdash \sigma \overset{type}{\longrightarrow} \tau}{\Gamma \vdash \sigma <\!\!\# \, \tau} \quad (Match{-}Red_L)$$

$$\frac{\Gamma \vdash \tau \overset{type}{\longrightarrow} \sigma}{\Gamma \vdash \sigma <\!\!\# \, \tau} \quad (Match{-}Red_R)$$

$$\frac{\Gamma, t <\!\!\# \, \sigma, \Gamma' \vdash ok}{\Gamma, t <\!\!\# \, \sigma, \Gamma' \vdash t <\!\!\# \, \sigma} \quad (Match{-}Var)$$

$$\frac{\Gamma \vdash \sigma <\!\!\# \, \tau \qquad \Gamma \vdash \tau <\!\!\# \, \rho}{\Gamma \vdash \sigma <\!\!\# \, \rho} \quad (Match{-}Trans)$$

$$\frac{\Gamma \vdash \sigma : T}{\Gamma \vdash \sigma <\!\!\# \, \sigma} \quad (Match{-}Refl)$$

$$\frac{\Gamma \vdash \tau <\!\!\# \, \tau' \qquad \Gamma \vdash \tau' \twoheadleftarrow \mathtt{m} : T}{\Gamma \vdash \tau \twoheadleftarrow \mathtt{m} <\!\!\# \, \tau' \twoheadleftarrow \mathtt{m}} \quad (Match{-}Extend)$$

Type-Reduction Rules

$$\frac{\Gamma \vdash \tau <\!\!\# \, \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma \triangleleft R' \rangle\!\rangle}{\Gamma \vdash \tau \twoheadleftarrow \mathtt{m} \overset{type}{\longrightarrow} \tau} \quad (Red{-}Over)$$

$(Red{-}Ext)$

$$\frac{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R \triangleleft R', \mathtt{m} : \sigma \rangle\!\rangle : T}{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R \triangleleft R', \mathtt{m} : \sigma \rangle\!\rangle \twoheadleftarrow \mathtt{m} \overset{type}{\longrightarrow} \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma \triangleleft R' \rangle\!\rangle}$$

$(Red{-}Pro)$

$$\frac{\Gamma, t <\!\!\# \, \mathtt{pro}\, t.\langle\!\langle R \triangleleft R' \rangle\!\rangle \vdash \sigma \overset{type}{\longrightarrow} \sigma'}{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma \triangleleft R' \rangle\!\rangle \overset{type}{\longrightarrow} \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma' \triangleleft R' \rangle\!\rangle}$$

$$\frac{\Gamma \vdash \tau \twoheadleftarrow \mathtt{m} : T \qquad \Gamma \vdash \tau \overset{type}{\longrightarrow} \tau'}{\Gamma \vdash \tau \twoheadleftarrow \mathtt{m} \overset{type}{\longrightarrow} \tau' \twoheadleftarrow \mathtt{m}} \quad (Red{-}Inherit)$$

$$\frac{\Gamma \vdash \sigma \overset{type}{\longrightarrow} \sigma' \qquad \Gamma \vdash \tau : T}{\Gamma \vdash \sigma \to \tau \overset{type}{\longrightarrow} \sigma' \to \tau} \quad (Red{-}Arrow_L)$$

$$\frac{\Gamma \vdash \tau \xrightarrow{type} \tau' \qquad \Gamma \vdash \sigma : T}{\Gamma \vdash \sigma \to \tau \xrightarrow{type} \sigma \to \tau'} \quad (Red{-}Arrow_R)$$

Type Rules for Lambda Terms

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

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

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

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

Type Rules for Object Terms

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

$(Pre{-}Extend)$

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

$$\frac{\begin{array}{l} \Gamma \vdash e_1 : \tau \\ \Gamma \vdash \tau <\!\#\, \mathtt{pro}\, t.\langle\!\langle R \lhd R', \mathtt{m} : \sigma \rangle\!\rangle \\ \Gamma, t <\!\#\, \tau \twoheadleftarrow \mathtt{m} \vdash e_2 : t \to \sigma \end{array}}{\Gamma \vdash \langle e_1 \longleftarrow \mathtt{m} = e_2 \rangle : \tau \twoheadleftarrow \mathtt{m}} \quad (Object)$$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash \tau <\!\#\, \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma \lhd R' \rangle\!\rangle}{\Gamma \vdash e \Leftarrow \mathtt{m} : \sigma[\tau/t]} \quad (Send)$$

$$\frac{\begin{array}{ll} \Gamma \vdash e_1 : \tau & \Gamma \vdash e_2 : \rho \\ \Gamma \vdash \tau <\!\#\, \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma \lhd R' \rangle\!\rangle & \Gamma \vdash \rho <\!\#\, \tau \end{array}}{\Gamma \vdash Sel(e_1, \mathtt{m}, e_2) : \sigma[\rho/t]} \quad (Select)$$

$$\frac{\Gamma \vdash e : \sigma \qquad \Gamma \vdash \sigma \xrightarrow{type} \tau}{\Gamma \vdash e : \tau} \quad (Red_L)$$

$$\frac{\Gamma \vdash e : \sigma \qquad \Gamma \vdash \tau \xrightarrow{type} \sigma}{\Gamma \vdash e : \tau} \quad (Red_R)$$

B The Extra Rules for Subsumption

Extra Well-formed Types

$(Type{-}Obj)$

$$\frac{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R \lhd R' \rangle\!\rangle : T \qquad t \text{ covariant in } R, R'}{\Gamma \vdash \mathtt{obj}\, t.\langle\!\langle R \lhd R' \rangle\!\rangle : T}$$

$$\frac{\Gamma \vdash \tau <\!\#\, \mathtt{obj}\, t.\langle\!\langle \lhd R, \mathtt{m} : \sigma \rangle\!\rangle}{\Gamma \vdash \tau \twoheadleftarrow \mathtt{m} : T} \quad (Type{-}Extend{-}Obj)$$

Extra Matching Rules

$$\frac{\Gamma \vdash \mathtt{obj}\, t.\langle\!\langle R \lhd R' \rangle\!\rangle : T}{\Gamma \vdash \mathtt{pro}\, t.\langle\!\langle R \lhd R' \rangle\!\rangle <\!\#\, \mathtt{obj}\, t.\langle\!\langle R \lhd R' \rangle\!\rangle} \quad (Promote)$$

$(Match{-}Book{-}Obj)$

$$\frac{\begin{array}{l} \Gamma \vdash \mathtt{obj}\, t.\langle\!\langle R \lhd R' \rangle\!\rangle : T \\ \Gamma \vdash \mathtt{obj}\, t.\langle\!\langle R \lhd R', \mathtt{m} : \sigma \rangle\!\rangle : T \end{array}}{\Gamma \vdash \mathtt{obj}\, t.\langle\!\langle R \lhd R', \mathtt{m} : \sigma \rangle\!\rangle <\!\#\, \mathtt{obj}\, t.\langle\!\langle R \lhd R' \rangle\!\rangle}$$

$$\frac{\Gamma \vdash \sigma' <\!\#\, \sigma \qquad \Gamma \vdash \tau <\!\#\, \tau'}{\Gamma \vdash \sigma \to \tau <\!\#\, \sigma' \to \tau'} \quad (Match{-}Arrow)$$

Extra Type-Reduction Rules

$$\frac{\Gamma \vdash \tau <\!\#\, \mathtt{obj}\, t.\langle\!\langle R, \mathtt{m} : \sigma \lhd R' \rangle\!\rangle}{\Gamma \vdash \tau \twoheadleftarrow \mathtt{m} \xrightarrow{type} \tau} \quad (Red{-}Over')$$

$(Red{-}Ext')$

$$\frac{\Gamma \vdash \mathtt{obj}\, t.\langle\!\langle R \lhd R', \mathtt{m} : \sigma \rangle\!\rangle : T}{\Gamma \vdash \mathtt{obj}\, t.\langle\!\langle R \lhd R', \mathtt{m} : \sigma \rangle\!\rangle \twoheadleftarrow \mathtt{m} \xrightarrow{type} \mathtt{obj}\, t.\langle\!\langle R, \mathtt{m} : \sigma \lhd R' \rangle\!\rangle}$$

$(Red{-}Obj)$

$$\frac{\Gamma, t <\!\#\, \mathtt{pro}\, t.\langle\!\langle R, \mathtt{m} : \sigma \lhd R' \rangle\!\rangle \vdash \sigma \xrightarrow{type} \sigma'}{\Gamma \vdash \mathtt{obj}\, t.\langle\!\langle R, \mathtt{m} : \sigma \lhd R' \rangle\!\rangle \xrightarrow{type} \mathtt{obj}\, t.\langle\!\langle R, \mathtt{m} : \sigma' \lhd R' \rangle\!\rangle}$$

New Type Rules for Expressions

$$\frac{\Gamma \vdash e : \sigma \qquad \Gamma \vdash \sigma <\!\#\, \tau \qquad \Gamma \vdash \tau : Rgd}{\Gamma \vdash e : \tau} \quad (Subsume)$$

$$\frac{\begin{array}{l} \Gamma \vdash e_1 : \tau \\ \Gamma \vdash \tau <\!\#\, \mathtt{obj}\, t.\langle\!\langle R \lhd R', \mathtt{m} : \sigma \rangle\!\rangle \\ \Gamma, t <\!\#\, \tau \twoheadleftarrow \mathtt{m} \vdash e_2 : t \to \sigma \end{array}}{\Gamma \vdash \langle e_1 \longleftarrow \mathtt{m} = e_2 \rangle : \tau \twoheadleftarrow \mathtt{m}} \quad (New{-}Object)$$

$(New{-}Send)$

$$\frac{\Gamma \vdash e : \tau \qquad \Gamma \vdash \tau <\!\#\, \mathtt{obj}\, t.\langle\!\langle R, \mathtt{m} : \sigma \lhd R' \rangle\!\rangle}{\Gamma \vdash e \Leftarrow \mathtt{m} : \sigma[\tau/t]}$$

$(New{-}Select)$

$$\frac{\begin{array}{ll} \Gamma \vdash e_1 : \tau & \Gamma \vdash e_2 : \rho \\ \Gamma \vdash \tau <\!\# \ \texttt{obj}\, t.\langle\!\langle R, \texttt{m} : \sigma \triangleleft R' \rangle\!\rangle & \Gamma \vdash \rho <\!\# \ \tau \end{array}}{\Gamma \vdash Sel(e_1, \texttt{m}, e_2) : \sigma[\rho/t]}$$

Rules for Fixed Types

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

$$\frac{\Gamma \vdash \texttt{obj}\, t.\langle\!\langle R \triangleleft R' \rangle\!\rangle : T}{\Gamma \vdash \texttt{obj}\, t.\langle\!\langle R \triangleleft R' \rangle\!\rangle : Rgd} \quad (Rgd{-}Obj)$$

$$\frac{\Gamma \vdash \tau : Rgd \qquad \Gamma \vdash \tau \twoheadleftarrow \texttt{m} : T}{\Gamma \vdash \tau \twoheadleftarrow \texttt{m} : Rgd} \quad (Rgd{-}Extend)$$

$$\frac{\Gamma \vdash \sigma : Rgd}{\Gamma, t <\!\# \ \sigma, \Gamma' \vdash t : Rgd} \quad (Rgd{-}Var)$$

$$\frac{\Gamma \vdash \sigma : Rgd \qquad \Gamma \vdash \tau : Rgd}{\Gamma \vdash \sigma \rightarrow \tau : Rgd} \quad (Rgd{-}Arrow)$$