Bounded Polymorphism for Extensible Objects

Luigi Liquori

Dipartimento di Matematica ed Informatica, Università di Udine,
Via delle Scienze 206, I-33100 Udine, Italy
e-mail: `liquori@dimi.uniud.it`
Current Address: LIP, École Normale Supérieure de Lyon
46, Allée d'Italie, F-69364 Lyon Cedex 07, France
e-mail: `Luigi.Liquori@ens-lyon.fr`

Abstract. In the ECOOP'97 conference, the author of the present paper investigated a conservative extension, called $\mathcal{O}b^{+}_{1<:}$, of the first-order Object Calculus $\mathcal{O}b_{1<:}$ of Abadi and Cardelli, supporting *method extension* in presence of *object subsumption*. In this paper, we extend that work with *explicit variance annotations* and *selftypes*. The resulting calculus, called $\mathcal{O}b^{+}_{s<:}$, is a proper extension of $\mathcal{O}b^{+}_{1<:}$. Moreover it is proved to be type sound.

Categories. Type systems, design and semantics of object-oriented languages.

1 Introduction

In the last few years, the problem of designing safe and expressive type-systems for object-based languages (also called prototype-based languages) has been widely addressed. The seminal works of [US87,CU89,Mic90,Aba94,FHM94,AC96a] share the same object-oriented philosophy, where the main entity is the one of *object* instead of the one of *class*. In those papers, classes can be easily codified by appropriate objects, following the "classes-as-objects" analogy of Smalltalk-80 [GR83]. In object-based languages, objects are modified directly from other objects (the latter called *prototypes*) by adding new methods, or by rewriting old method bodies with new ones. A primitive operation of method call is given, to send a message to (i.e. invoke a method on) an object. In functional calculi, adding or rewriting a method produces a new object that inherits all the properties of the original one.

Another key issue in object-based languages is the one of *subsumption*, i.e. the capability to use an object with a longer (or more refined) interface in every context expecting objects with a smaller (or less refined) interface. This feature has been showed to be fundamental in object-oriented paradigm, since it allows a significant reuse of code. Unfortunately, as clearly stated in [FM94,AC96a], adding object subsumption in presence of object extension make the type system very often unsound.

As a simple example of this problem, let us suppose to have a diagonal point `dpoint` composed by two fields, `x` (holds 1) and `y` (holds `self.x`). The type of this object is [`x`:nat, `y`:nat]. If we "hide", by subsumption, the `x` field, and we add again `x` with a new value -1 of type int, and we call `y` on the object `dpoint`, then we lose the subject reduction property, since the evaluation of $dpoint.y$, of type nat, yields the value -1 of type int. Other works by [FM95,BL95,Rém95,BBDL97,Rém98,RS98], have addressed the issue of integrating object subsumption in presence of object extension.

This paper starts from the Abadi & Cardelli's (first-order) Object Calculus, called $\mathcal{O}b_{1<:}$ [AC96b]. We briefly recall its features.

- it supports "fixed size" objects (no object extension is provided);
- it supports method override;
- it supports object subsumption;
- its type system catches run-time errors such as *message-not-understood*.

In [Liq97b], the $\mathcal{O}b_{1<:}$ calculus was extended by allowing object extension compatible with object subsumption, by providing a sound static type system and a typed equational theory on objects. This (conservative) extension was called $\mathcal{O}b^{+}_{1<:}$. This paper completes the work of [Liq97b] by extending the type system of $\mathcal{O}b^{+}_{1<:}$ with selftypes and explicit variance annotations.

Selftypes has been showed to be fruitful in a development of flexible type-systems for object oriented programming languages (e.g. Eiffel [Mey92], PolyTOIL [BSvG95]). Selftypes allow one to give a type to methods that return `self` or an update of `self` (for instance, a `move` method of a `point` object will have type $int \rightarrow$`selftype`, where `selftype` refers to the type of `self`). Adding selftypes to object-calculi is not only an exercise of style: in fact we can give a type to a considerably number of programs that are not typable within the first-order fragment of $\mathcal{O}b^{+}_{1<:}$.

Explicit variance annotations, instead, support flexible subtyping, and a direct protection tool from unwanted "read" or "write" operations. More precisely, an explicit variance annotation is a "label" attached to a method name and defined together with the method body; it could be one of the following: `private`, `public`, `read_only`, and `write_only`. The meaning of explicit variance annotations is straightforward: they denote the access privileges of fields/methods belonging to the object. Having explicit variance annotations inside the calculus allows a more disciplined use of methods and fields, and enforces object encapsulation.

The addition of selftypes fits well into the type system of [Liq97b], where we distinguish between two "kinds" of objects-types, namely the *saturated* object-types, and the *diamond* object-types. Shortly, if an object can be typed by a saturated object-type, then it can receive messages and override the methods that it contains. Instead, if an object can be typed by a diamond object-type, then it can receive messages, override some methods, and it can be extended by new methods. On both types, a subtyping relation is defined.

The subtyping relation on saturated object-types can be commonly found in the literature: at first approximation, an object typed with a "longer" (i.e. with more methods) object-type can be used in any context expecting an object typed with a "shorter" (i.e. with less methods) object-type. At this level, object extension is forbidden since we can first "hide", by subsumption, a method `m` of type σ, and then extend the object with the same method `m` of type τ, σ being *incompatible* with τ.

For diamond object-types, instead, the subtype relation behaves as follows: it is still possible to hide a method, but its type is *recorded* in the diamond object-type. Since object extension is only allowed on objects typed with diamond object-types, the hidden methods can be re-added again only with the *same* type.

The $\mathcal{O}b^{+}_{s<:}$ calculus that we present in this paper is a conservative extension of the first-order one $\mathcal{O}b^{+}_{1<:}$. In summary, our calculus exhibits the following features:

– extendible objects with appropriate method specialization of inherited methods,
– a (*mytype-covariant*) subtyping relation compatible with object extension,
– explicit variance annotations;
– override of explicit variance annotations;
– static detection of run-time errors, such as *message-not-understood*.

This paper is organized as follows: in Section 2 we will present the Extended Object Calculus à la Curry (i.e. without type decorations). In Section 3 we will introduce the types, decorate our $\mathcal{O}b^{+}_{s<:}$ calculus with types, and present the type system. A number of examples which are meant to give an insight of the power of $\mathcal{O}b^{+}_{s<:}$ will be provided in Section 4. The last section will be devoted to a comparison with the paper of Abadi and Cardelli [AC95], the paper of Didier Rémy [Rém98], and the paper of Riecke and Stone [RS98]. Part of this material appeared in two technical reports [Liq97a], and [Liq99].

Acknowledgement. The author is grateful the anonymous referees to their helpful comments on this work.

2 The Extended Primitive Object Calculus

The untyped syntax of the Extended Object Calculus is defined by the following grammar:

$$\begin{array}{rcl} \mathrm{o} & ::= & s \mid [\mathrm{m}_i \Upsilon_i = \varsigma(s_i)\mathrm{o}_i]^{i \in I} \mid \mathrm{o.m} \mid \mathrm{o.m} := \varsigma(s)\mathrm{o} \mid \mathrm{o.m} := \Upsilon \mid \mathrm{o.m} := \Upsilon\varsigma(s)\mathrm{o} \\ \Upsilon & ::= & \mathtt{private} \mid \mathtt{public} \mid \mathtt{read\_only} \mid \mathtt{write\_only}. \end{array}$$

Here the $:=$ operator can be intended as an operator on objects which overrides method $\mathtt{m}$ in case this method is already present in the object, otherwise it extends the object with $\mathtt{m}$. The grammar for Υ denotes *explicit variance annotations* that are introduced to support a clear form of encapsulation and protection from unwanted "read" or "write" operations. The expression $\mathtt{o.m} := \Upsilon$ modifies (i.e. overrides) the explicit variance annotation for $\mathtt{m}$. The explicit variance annotations have the following intuitive meaning:

– `public`: methods that have both read/write privilege;
– `read_only`: methods that only have read privilege;
– `write_only`: methods that only have write privilege;
– `private`: methods that do not have read/write privilege, i.e. "encapsulated".

2.1 Small-step Operational Semantics

Let $\mathrm{o}\{s\}$ denote an object where the variable s can freely occur, let $\mathrm{o}\{\mathrm{o}'\}$ denote the substitution of the object o' for every free occurrence of s in o when $\mathrm{o}\{s\}$ is present in the same context, and let, for $i, j \in I$, with $i \neq j$, $\mathtt{m}_i$ and $\mathtt{m}_j$ be distinct methods. The

Let $\mathsf{o} \triangleq [\mathtt{m}_i \Upsilon_i = \varsigma(s_i)\mathsf{o}_i\{s_i\}]^{i\in I}$						
(Sel)	$\mathsf{o}.\mathtt{m}_j$	$\stackrel{ev}{\to}$	$\mathsf{o}_j\{\mathsf{o}\}$	$(j \in I)$	(a)	(1)
$(Over)$	$\mathsf{o}.\mathtt{m}_j := \varsigma(s_j)\mathsf{o}'$	$\stackrel{ev}{\to}$	$[\mathtt{m}_i \Upsilon_i = \varsigma(s_i)\mathsf{o}_i, \mathtt{m}_j \Upsilon_j = \varsigma(s_j)\mathsf{o}']^{i\in I\setminus\{j\}}$	$(j \in I)$	(b)	(2)
(Ann)	$\mathsf{o}.\mathtt{m}_j := \Upsilon$	$\stackrel{ev}{\to}$	$[\mathtt{m}_i \Upsilon_i = \varsigma(s_i)\mathsf{o}_i, \mathtt{m}_j \Upsilon = \varsigma(s_j)\mathsf{o}_j]^{i\in I\setminus\{j\}}$	$(j \in I)$	(c)	(3)
(Ext)	$\mathsf{o}.\mathtt{m}_j := \Upsilon\varsigma(s_j)\mathsf{o}'$	$\stackrel{ev}{\to}$	$[\mathtt{m}_i \Upsilon_i = \varsigma(s_i)\mathsf{o}_i, \mathtt{m}_j \Upsilon = \varsigma(s_j)\mathsf{o}']^{i\in I}$	$(j \notin I)$		(4)

Table 1. Small-step Untyped Operational Semantics

small-step operational semantics can be given as the reflexive, transitive and contextual closure of the reduction relation defined in Table 1. Note that the original semantics of [AC96a] was build from the reduction rules (1) and (2). As usual, we do not make error conditions explicit. Let $\stackrel{ev}{\twoheadrightarrow}$ be the general many-step reduction. We remark that the (Ann) rule overrides the explicit variance annotation, leaving the method body unchanged; orthogonally, the $(Over)$ rule modifies the method body, leaving the explicit method annotation unchanged. The condition $(a), (b), (c)$ are the following ones:

$$
\begin{aligned}
(a) &\triangleq \Upsilon_j \in \{\mathtt{public}, \mathtt{read\_only}\} \\
(b) &\triangleq \Upsilon_j \in \{\mathtt{public}, \mathtt{write\_only}\} \\
(c) &\triangleq \Upsilon_j : \upsilon_j,\ \ \Upsilon : \upsilon,\ \text{ and } \upsilon_j <: \upsilon.
\end{aligned}
$$

The condition (a) allows message selection only for fields/methods that are public or readable from the outside (i.e. annotated with `public`, or `read_only`). The condition (b) allows overriding only for fields/methods that are public or writable from the outside (i.e. annotated with `public`, or `write_only`). The condition (c) can be explained as follows. A variance annotation (or *variance type* υ) can be assigned to an explicit variance annotation (Υ) via a simple "type" system proving judgments of the shape $\Upsilon : \upsilon$, where $\upsilon \in \{^+, ^-, ^\circ, ^\bullet\}$. The type rules are:

$$\mathtt{public} : {}^\circ \quad \mathtt{private} : {}^\bullet \quad \mathtt{read\_only} : {}^+ \quad \mathtt{write\_only} : {}^-.$$

Given that, the (c) condition assures that the new explicit variance annotation Υ will override the original one Υ_j only if their variance types are compatible. Compatibility is assured by a partial order relation $(<:)$ on variance types, given by the following "chains":

$${}^\circ <: {}^+ <: {}^\bullet, \quad \text{and} \quad {}^\circ <: {}^- <: {}^\bullet.$$

As a remark, we observe that we could, in principle, build a simpler and more liberal small-step semantics by dropping the side conditions (a), (b), and (c). The type system always guarantees the soundness of well-typed expressions.

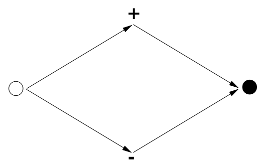
For the small-step operational semantics, we can derive an untyped equational theory (whose judgment is $\vdash \mathsf{o} \stackrel{ev}{=} \mathsf{o}'$) from the reduction rules, by simply adding rules for symmetry, transitivity and congruence, and reformulating the reduction rules as equalities. We can also define quite simply a big-step operational semantics that also induces a "lazy" strategy of evaluation, via a natural proof deduction system à la Plotkin. This semantics maps every closed expression into a normal form, i.e. an irreducible term (for a presentation of the big-step semantics and of the equational theory see [Liq99]).

σ, τ ::=		
	t, u	type-variables
	ω	the biggest type
	$\mathtt{obj}\, t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\}]^{i \in I}$	*saturated* object-type, $\mathtt{m}_i$ distinct
	$\mathtt{obj}\, t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\} \diamond \mathtt{m}_j \upsilon_j : \sigma_j\{t\}]^{i \in I}_{j \in J}$	*diamond* object-type, $\upsilon_j \in \{^\circ, ^-\}, I \cap J = \emptyset$

Table 2. Syntax of Types

3 The Type System

In the $\mathcal{O}b^{+}_{s<:}$ type system, the set of legal types is defined by the grammar of Table 2. The type-constant ω is the supertype of every type. We omit how to encode basic data-types which can be treated as in [AC96a]. The bound type-variable t can (freely) occur in the σ_i, σ_j's, and it is constrained to be *covariant*. As explained in many papers, (among others [Cas95,Cas96,BCC+96,AC96a,Liq98]) the covariance of `selftype` is necessary if we want to have a statically typed calculus with subtyping. As such, *binary methods* (i.e. methods that receive as input an argument of the same type of `self`) are lost. When a method $\mathtt{m}_j$ ($j \in I$) is invoked, the result will have a type $\sigma_j\{t\}$ in which every free occurrence of t is replaced with the type τ of the receiver of the message, i.e. $\sigma_j\{\tau\}$, therefore showing the "recursive" nature of that type.
Explicit Variance Annotations. As we have sketched in the previous section, each υ_i, υ_j inside object-types is a variance annotation, i.e. one of the symbols $^+$, $^-$, $^\circ$, or $^\bullet$, standing, respectively, for *covariance*, *contravariance*, *public-invariance*, and *private-invariance*. Any omitted υ's are taken to be equal to $^\circ$. Covariant methods allow covariant subtyping, but prevent update (see [FM94,AC96a]). Symmetrically, contravariant methods allow contravariant subtyping, but prevent invocation. Public-invariant methods, instead, can be invoked and updated. By subtyping, public-invariant methods can be regarded as either covariant or contravariant. Private-invariant methods cannot be invoked nor updated: these methods are typically introduced (and hence type-checked) being public, or readable, or writable, but are later "sealed" (implicitly via subtyping, or explicitly via annotation override) as private methods that cannot be accessed nor updated from the outside. The "compatibility" relation between variance annotations is depicted below (where $\upsilon \rightarrow \upsilon'$ means $\upsilon <: \upsilon'$, i.e. a method annotated with υ can be also annotated with υ'), together with all possible forms of protection from the outside of the object performed by variance annotations.



Variance	Read?	Write?
$\circ$	Yes	Yes
$+$	Yes	No
$-$	No	Yes
$\bullet$	No	No

Saturated-types. The saturated-types $\mathtt{obj}\, t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\}]^{i \in I}$ are the ordinary object-types of [AC96a]; shortly, objects assigned to saturated-types can receive messages and can be rewritten.
Diamond-types. The diamond-types $\mathtt{obj}\, t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\} \diamond \mathtt{m}_j \upsilon_j : \sigma_j\{t\}]^{i \in I}_{j \in J}$ are directly derived from the one of [Liq97b]. Diamond-types can be assigned to objects

which can be extended and overridden. The symbol $\diamond$ distinguishes the two parts of that object-type, i.e. the *interface-part* and the *subsumption-part*; the former part describes all methods (with their types) that *may be invoked* (if not private or write-only), the latter conveys, instead, information about (the types of) methods that are subsumed in the type-checking phase. When a method is subsumed in a diamond-type it simply moves from the interface-part to the subsumption-part. This "shift" guarantees that any future addition of that method will be type-consistent with the previous one. The subsumption-part is also used as a infinite "container" of unused method types; this is important when we need to add a "fresh" method, in order to not loose the full flexibility of rapid prototyping. The shifting and the stocking of methods are performed using a suitable subtype system, presented in the Appendix.

Variance annotations are elegantly integrated within object-types. Since a method can also "migrate" from the subsumption-part to the interface-part by object extension, and since subsumed methods cannot be invoked, it follows that the occurrence of $\mathtt{m}\upsilon : \sigma$ in the subsumption-part of a diamond-type is allowed only if $\upsilon \in \{^\circ, ^-\}$, i.e. for public or write-only methods (an object extension of a previously subsumed method behaves, operationally, as an object override).

3.1 Types and Judgments

The judgments we set about to prove have the forms:

$$\Gamma \vdash ok, \quad \Gamma \vdash \sigma, \quad \Gamma \vdash \circ : \sigma, \quad \Gamma \vdash \sigma <: \tau, \quad \Gamma \vdash \upsilon\sigma <: \upsilon\tau,$$

where Γ is a context which gives meaning to the free variables of $\circ$, σ, and τ, generated by the grammar: $\Gamma ::= \varepsilon \mid \Gamma, s : \sigma \mid \Gamma, u <: \sigma$. In contexts, we often write $s : u <: \sigma$, to denote $u <: \sigma, s : u$. By deriving the first two judgments we check the well-formation of the context Γ and of the type σ, respectively; while with the third one, we assign a type σ to the expression $\circ$. The last two judgments are the usual subtyping judgments between types (with variance annotations) of [AC96a]. As shown in Section 2, in order to override an explicit method annotation, we need the auxiliary judgment $\Upsilon : \upsilon$, that assigns a variance type υ to an explicit variance annotation Υ.

Cova/Contravariance. Formally, $\sigma\{t^+\}$ stands for a type where the type-variable t occurs only covariantly. Intuitively, $\sigma\{u^+\}$ means that u occurs *at most* positively in σ; similarly, $\sigma\{u^-\}$ means that u occurs *at most* negatively in σ. The formal definition of covariance follows in Table 3.

The type rules for well-formed contexts and types are routine, and can be found in Appendix. We only remark that in the $(T{-}\diamond)$ rule, we require that, for all $j \in J$, the type annotations υ_j, must belong to $\{^\circ, ^-\}$, so allowing a method to be "writable".

3.2 Subtyping

The more important subtyping rules are presented in Table 4; the full set can be found in Appendix. The subtyping rules that deal with diamond-types and variance types are the same as in [Liq97b], and [AC95], respectively (see Appendix). Moreover we need some extra rules, for instance the rules $(S{-}Var_\diamond)$ and $(S{-}Var)$ to deal with variance

Covariance	
$t\{u^+\}$	always
$\omega\{u^+\}$	always
$\mathtt{obj}\, t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\}]^{i\in I}\{u^+\}$	if $t = u$ or for all $i \in I$: $\begin{cases} \text{if } \upsilon_i \equiv^+\text{, then } \sigma_i\{u^+\} \\ \text{if } \upsilon_i \equiv^-\text{, then } \sigma_i\{u^-\} \\ \text{if } \upsilon_i \equiv^\circ\text{, then } u \notin FV(\sigma_i) \\ \text{if } \upsilon_i \equiv^\bullet\text{, always} \end{cases}$
Contravariance	
$t\{u^-\}$	if $t \neq u$
$\omega\{u^-\}$	always
$\mathtt{obj}\, t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\}]^{i\in I}\{u^-\}$	if $t = u$ or for all $i \in I$: $\begin{cases} \text{if } \upsilon_i \equiv^+\text{, then } \sigma_i\{u^-\} \\ \text{if } \upsilon_i \equiv^-\text{, then } \sigma_i\{u^+\} \\ \text{if } \upsilon_i \equiv^\circ\text{, then } u \notin FV(\sigma_i) \\ \text{if } \upsilon_i \equiv^\bullet\text{, always} \end{cases}$
Private/Public Invariance	
$\sigma\{u^\bullet\}$	if $\sigma\{u^+\}$ or $\sigma\{u^-\}$
$\sigma\{u^\circ\}$	if neither $\sigma\{u^+\}$ nor $\sigma\{u^-\}$ nor $\sigma\{u^\bullet\}$
Variance & $\diamond$-types	
$\mathtt{obj}\, t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\}\diamond \mathtt{m}_j \upsilon_j : \sigma_j\{t\}]^{i\in I}_{j\in J}\{u^v\}$	if $\mathtt{obj}\, t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\}]^{i\in I}\{u^v\}$ and $\mathtt{obj}\, t.[\mathtt{m}_j \upsilon_j : \sigma_j]^{j\in J}\{u^v\}$

Table 3. Variance Occurrences

types for object-types of the same length, and the rule $(S{-}Inv_2)$ to say that a read-only or write-only component can be regarded as a private one. The rule $(S{-}Inv_1)$ is simply a reformulation of reflexivity. As a side remark, observe that the condition $\forall\ k \in I \cup J$ in rule $(S{-}Var_\diamond)$ allows to apply this rule also in the subsumption-part of the diamond-type. This condition is more liberal than the simpler $\forall\ k \in I$, since it allows one to re-add a forgotten method with a type different from the one we have forgotten (in accordance to its variance type), without losing type soundness.

3.3 Type Rules

We decorate our Extended Object Calculus with types as follows:

$$\begin{aligned} \mathtt{o}\ ::=\ & s \mid [\mathtt{m}_i \Upsilon_i = \varsigma(s_i{:}u <: \tau_i)\mathtt{o}_i]^{i\in I} \mid \mathtt{o.m} \mid \mathtt{o.m} := \varsigma(s{:}u <: \tau)\mathtt{o} \mid \\ & \mathtt{o.m} := \Upsilon \mid \mathtt{o.m} := \Upsilon\varsigma(s{:}u <: \tau)\mathtt{o}. \end{aligned}$$

The ς-binder scopes over the object-variable s, referring to `self`, and the type-variable u, referring to the type of `self` (i.e. `selftype`). The method bodies could be intended, in the $F_{<:}$ jargon, as the polymorphic lambda abstraction $\Lambda u <: \sigma_i.\lambda s{:}u.\mathtt{o}_i$. We analyze in detail the most important type rules of $\mathcal{O}b^{+}_{s<:}$(presented in Table 5); see Appendix for the full set of rules.

[$(V{-}Sel)$] This rule gives a type for a message send; in order for a message send to be type correct, the host object $\mathtt{o}$ must contain the method name $\mathtt{m}_k$ in its type. Moreover, the substitution of t with τ reflects the recursive nature of object-types. The host object $\mathtt{o}$ can also be an object-variable s: in this case the type τ will be a type-variable u. Method selection is permitted only on public-invariant or covariant components.

$$(S\!-\!Var_\diamond)\quad \frac{\Gamma, u <: \mathtt{obj}\, t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\} \diamond \mathtt{m}_j \upsilon_j : \sigma_j\{t\}]^{i\in I}_{j\in J} \vdash \upsilon_k \sigma_k\{u\} <: \upsilon'_k \sigma'_k\{u\} \quad \forall\, k \in I \cup J}{\Gamma \vdash \mathtt{obj}\, t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\} \diamond \mathtt{m}_j \upsilon_j : \sigma_j\{t\}]^{i\in I}_{j\in J} <: \mathtt{obj}\, t.[\mathtt{m}_i \upsilon'_i : \sigma'_i\{t\} \diamond \mathtt{m}_j \upsilon'_j : \sigma'_j\{t\}]^{i\in I}_{j\in J}}$$

$$(S\!-\!Var)\quad \frac{\Gamma, u <: \mathtt{obj}\, t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\}]^{\,i\in I} \vdash \upsilon_k \sigma_k\{u\} <: \upsilon'_k \sigma'_k\{u\} \quad \forall\, k \in I}{\Gamma \vdash \mathtt{obj}\, t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\}]^{\,i\in I} <: \mathtt{obj}\, t.[\mathtt{m}_i \upsilon'_i : \sigma'_i\{t\}]^{\,i\in I}}$$

$$\frac{\Gamma \vdash \sigma \quad \upsilon \in \{^\circ, ^\bullet\}}{\Gamma \vdash \upsilon\sigma <: \upsilon\sigma}\ (S\!-\!Inv_1) \qquad \frac{\Gamma \vdash \sigma \quad \upsilon \in \{^+, ^-\}}{\Gamma \vdash \upsilon\sigma <: {}^\bullet\sigma}\ (S\!-\!Inv_2)$$

Table 4. Some Subtyping Rules

[$(V\!-\!Over)$] This rule overrides the method $\mathtt{m}_k$ provided that $\mathtt{m}_k$ belongs to the interface of the object o, (i.e. $k \in I$), and that the new body for $\mathtt{m}_k$ uses the methods already present in o; this last condition is ensured by the second subtyping judgment of the premises, and corresponds to say that those methods are present in the interface-part of the type τ. Object override is allowed only on public-invariant or contravariant components. We also observe that τ can also be a type-variable, and, as such, method override is allowed inside method bodies.

[$(V\!-\!Ann_1)$] This rule overrides the explicit variance annotation for method $\mathtt{m}_k$ (already present in the type of o), only if the new annotation Υ has a variance type compatible with the variance type of $\mathtt{m}_k$ present in the object-type assigned to o. In this rule, the type of the object o is a saturated-type but can be a diamond-type as well, as in rule $(V\!-\!Ann_2)$. The second premise guarantees the presence of method $\mathtt{m}$ and the compatibility of its variance type with the new one.

[$(V\!-\!Ext)$] This rule extends an object o with a method $\mathtt{m}_k$. Firstly, one can see that we cannot extend an object whose object-type is saturated. Secondly, this rule extends an object with a new (fresh) method if and only if that method is present in the subsumption-part of the diamond-type assigned to the object to be extended. But this condition can always be satisfied by a diamond-type thanks to the subtyping rule $(S\!-\!Ext_\diamond)$. Of course we have $\Upsilon : \upsilon_k$. The condition $H \subseteq I$ guarantees that the methods which are essential to type the body o' are already present in the interface-part of the type $\mathtt{obj}\, t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\} \diamond \mathtt{m}_j \upsilon_j : \sigma_j\{t\}]^{i\in I}_{j\in J}$.

Note that this rule can also be applied when the method belongs to o but has been already subsumed via an application of a subtyping rule $(S\!-\!Shift_\diamond)$. In this case, operationally, is a method override. Moreover observe that, since object extension modifies from the outside the object, it follows that we can extend an object only with public or write only components. In fact, by looking at the subtyping rules, we can see that all variance annotations inside the subsumption-part are public-invariant or contravariant. As minor remarks on object extension, observe that:

– a "self-extension" operation is forbidden inside method bodies: in other words, the object $\mathrm{o} \triangleq [\mathtt{m} = \varsigma(s)s.\mathtt{n} := \varsigma(s)1]$, where $\mathtt{n}$ does not belong to o, cannot be type-decorated, because we are not able to give any correct type for the method $\mathtt{m}$.

$$\frac{\Gamma \vdash \mathtt{o} : \tau \qquad \Gamma \vdash \tau <: \mathtt{obj}\, t.[\mathtt{m}_k \upsilon_k : \sigma_k\{t\}] \qquad \upsilon_k \in \{^\circ, ^+\}}{\Gamma \vdash \mathtt{o}.\mathtt{m}_k : \sigma_k\{\tau\}} \quad (V{-}Sel)$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathtt{o} : \tau \qquad \Gamma \vdash \tau <: \mathtt{obj}\, t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\}]^{i \in I} \qquad k \in I \\ \Gamma, s_k : u <: \mathtt{obj}\, t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\}]^{i \in I} \vdash \mathtt{o}' : \sigma_k\{u\} \qquad \upsilon_k \in \{^\circ, ^-\}\end{array}}{\Gamma \vdash \mathtt{o}.\mathtt{m}_k := \varsigma(s_k{:}u <: \mathtt{obj}\, t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\}]^{i \in I})\mathtt{o}' : \tau} \quad (V{-}Over)$$

$$\frac{\begin{array}{l}\Gamma \vdash \mathtt{o} : \mathtt{obj}\, t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\}]^{i \in I} \\ \Gamma \vdash \mathtt{obj}\, t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\}]^{i \in I} <: \mathtt{obj}\, t.[\mathtt{m}_k \upsilon : \sigma_k\{t\}] \qquad \Upsilon : \upsilon\end{array}}{\Gamma \vdash \mathtt{o}.\mathtt{m}_k := \Upsilon : \mathtt{obj}\, t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\}, \mathtt{m}_k \upsilon : \sigma_k\{t\}]^{i \in I \setminus \{k\}}} \quad (V{-}Ann_1)$$

(Let $\tau_k \triangleq \mathtt{obj}\, t.[\mathtt{m}_h \upsilon_h : \sigma_h\{t\}]^{h \in H \cup \{k\}}$).

$$\frac{\begin{array}{l}\Gamma \vdash \mathtt{o} : \mathtt{obj}\, t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\} \diamond \mathtt{m}_j \upsilon_j : \sigma_j\{t\}]^{i \in I}_{j \in J} \qquad k \in J \\ \Gamma, s_k : u <: \tau_k \vdash \mathtt{o}' : \sigma_k\{u\} \qquad \Upsilon : \upsilon_k \qquad H \subseteq I\end{array}}{\Gamma \vdash \mathtt{o}.\mathtt{m}_k := \Upsilon\varsigma(s_k{:}u <: \tau_k)\mathtt{o}' : \mathtt{obj}\, t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\} \diamond \mathtt{m}_j \upsilon_j : \sigma_j\{t\}]^{i \in I \cup \{k\}}_{j \in J \setminus \{k\}}} \quad (V{-}Ext)$$

Table 5. Some Term Typing Judgments

- inside method bodies, the ς-bound variables s_i (referring to `self`) in the same object $\mathtt{o}$ have different bound object-types. As an example consider the object $[\mathtt{m} = \varsigma(s{:}u <: [\mathtt{m}{:}int])1, \mathtt{n} = \varsigma(s'{:}u <: [\mathtt{m}{:}int, \mathtt{n}{:}int])s'.\mathtt{m}]$ of type $[\mathtt{m}{:}int, \mathtt{n}{:}int]$. This fits well with the semantics of the message send thanks to the presence of the subtyping rule $(S{-}Width)$.
- if we override the method $\mathtt{n}$ of $\mathtt{o}'$ with a new body (e.g. $\varsigma(s{:}u <: [\mathtt{n}{:}int])1$), the new bound for u in $\mathtt{n}$ does not need to be related with the older one; this is sound because the bound depends on the methods useful to type the new body.
- thanks to our sophisticated subtyping system we are not obliged to know "a priori" (in advance) all the future extensions of an object; in fact, the saturated-part of a diamond-type can always be filled with fresh methods thanks to the rule $(S{-}Ext_\diamond)$.

The type system enjoy the subject reduction property.

Theorem 1 (Subject Reduction for $\mathcal{O}b^{+}_{s<:}$).
If $\Gamma \vdash \mathtt{o} : \sigma$ and $\mathtt{o} \overset{ev}{\to} \mathtt{o}'$, then $\Gamma \vdash \mathtt{o}' : \sigma$.

4 Applications

In this section, we present a number of examples that help to illustrate the features of $\mathcal{O}b^{+}_{s<:}$. Any unspecified Υ and υ are taken to be equal to `public` and $^\circ$ respectively.

Method Specialization. The following extendible point

$$\mathtt{point} \triangleq [\mathtt{x} = \varsigma(s{:}u <: \sigma_1)1,\ \mathtt{plus1} = \varsigma(s{:}u <: \sigma_2)s.\mathtt{x} := \varsigma(s'{:}u' <: \sigma_1)s.\mathtt{x} + 1],$$

is typable with $\mathtt{obj}\, t.[\mathtt{x}{:}int, \mathtt{plus1}{:}t\, \diamond]$, being $\sigma_1 \equiv [\mathtt{x}{:}int]$, and $\sigma_2 \equiv \mathtt{obj}\, t.[\mathtt{x}{:}int, \mathtt{plus1}{:}t]$.

Subtyping. Let `point` be as before, and let `c_point` be obtained by extending `point` with a `col` field. By an inspection of the typing rules for $\mathcal{O}b^{+}_{s<:}$ we derive $\vdash$ `point` : $P_\diamond$, and $\vdash$ `c_point` : $CP_\diamond$, where

$$\begin{array}{lcllcl} P & \triangleq & \mathtt{obj}\,t.[\mathtt{x}{:}int, \mathtt{plus1}{:}t] & CP & \triangleq & \mathtt{obj}\,t.[\mathtt{x}{:}int, \mathtt{col}{:}colors, \mathtt{plus1}{:}t] \\ P_\diamond & \triangleq & \mathtt{obj}\,t.[\mathtt{x}{:}int, \mathtt{plus1}{:}t\,\diamond] & CP_\diamond & \triangleq & \mathtt{obj}\,t.[\mathtt{x}{:}int, \mathtt{col}{:}colors, \mathtt{plus1}{:}t\,\diamond]. \end{array}$$

Now consider the following programs and related (derivable) types, where we introduce λ-binders to denote functions:

$$\begin{array}{lcllcl} f_1 & \triangleq & \lambda(s{:}P)s.\mathtt{x} & : & P{\to}int \\ f_2 & \triangleq & \lambda(s{:}P)s.\mathtt{x} := \varsigma(s'{:}u <: [\mathtt{x}{:}int])2 & : & P{\to}P \\ f_3 & \triangleq & \lambda(s{:}P_\diamond)s.\mathtt{col} := \varsigma(s'{:}u <: [\mathtt{col}{:}colors])\mathtt{red} & : & P_\diamond{\to}CP_\diamond. \end{array}$$

Again, by inspecting the typing rules, we find that the following judgments are derivable:

$$\begin{array}{llllll} \vdash & f_1\,(\mathtt{point}) & : int & \vdash & f_1\,(\mathtt{c\_point}) & : int \\ \vdash & f_2\,(\mathtt{point}) & : P & \vdash & f_2\,(\mathtt{c\_point}) & : P \\ \vdash & f_3\,(\mathtt{point}) & : CP_\diamond & (\nvdash & f_3\,(\mathtt{c\_point}) & : CP_\diamond). \end{array}$$

The last judgment is correctly false since $CP_\diamond \not<: P_\diamond$.

Method Annotations for Encapsulation. Consider an object `p` with a field `x` and two methods, namely `set` and `get`, invokable from the outside which, respectively, return and modify the value of `x`. It is natural to give the following saturated-type to `p`:

$$Point \triangleq \mathtt{obj}\,t.[\mathtt{x}^\circ : int, \mathtt{get}^\circ : int, \mathtt{set}^\circ : int{\to}t].$$

Then, in order to make the local field `x` protected against external access, and the `get` and `set` methods not writable, we could override `p` as follow:

$$\mathtt{prot\_p} \triangleq ((\mathtt{p.x} := \mathtt{private}).\mathtt{get} := \mathtt{read\_only}).\mathtt{set} := \mathtt{read\_only},$$

of type

$$ProtPoint \triangleq \mathtt{obj}\,t.[\mathtt{x}^\bullet : int, \mathtt{get}^+ : int, \mathtt{set}^+ : int{\to}t],$$

being that $Point <: ProtPoint$. So, the `x` variable becomes protected from the outside, and the `get` and `set` methods can be only invoked but not updated. As such, we obtain a neat distinction between public messages (i.e. the interface visible outside the object) and private variables (i.e. variables or local methods not accessible from the outside).

Classes as Collection of Pre-methods. In [Liq97b] a first-order encoding of classes-as-objects was given. As the $\mathcal{O}b^{+}_{s<:}$ is an extension of [Liq97b], it clearly follows that it also permit the building of classes and class instances. However, other encoding of classes are possible, provided that we increase our $\mathcal{O}b^{+}_{s<:}$ with polymorphic types. By polymorphic types we are able to build classes as a collection of parametric pre-methods[1]. A "pre-methods" is a polymorphic procedure that can be later used to construct a method parametric in the type of `self`. As an example, let the following object $\texttt{mem} \triangleq [\texttt{get} = \varsigma(s)\texttt{true}, \texttt{set} = \varsigma(s)\lambda(b)s.\texttt{get}{:=}\varsigma(s')b]$ of type $Mem \triangleq \texttt{obj}\,t.[\texttt{get} : \mathit{bool}, \texttt{set} : \mathit{bool}{\to}t]$, and consider the "class" `memClass` of [AC95] (for the sake of simplicity, all type-decorations are omitted, and $\lambda(\ )$ stands for polymorphic type-abstraction)

$$\begin{aligned} \texttt{memClass} \;\triangleq\; & [\texttt{new} = \varsigma(s)[\texttt{get} = \varsigma(s')s.\texttt{pre-get}(\ )(s'), \\ & \qquad\qquad\quad \texttt{set} = \varsigma(s')s.\texttt{pre-set}(\ )(s')], \\ & \;\texttt{pre-get} = \varsigma(s)\lambda(\ )\lambda(s')\texttt{false} \\ & \;\texttt{pre-set} = \varsigma(s)\lambda(\ )\lambda(s')\lambda(b)s'.\texttt{get} := \varsigma(s'')b], \end{aligned}$$

of type $Class(Mem) \triangleq [\texttt{new} : Mem, \texttt{pre-get}{:}\forall(u <: Mem)u{\to}\mathit{bool}, \texttt{pre-set}{:}$ $\forall(u <: Mem)u{\to}\mathit{bool}{\to}u]$. The `pre-get` and `pre-set` methods of `memClass` are parametric pre-methods that do not use the `self` of `memClass`; they are used inside the bodies of `get` and `set` of the class instances generated by the `new` method of `memClass`. An instance `mem` of `memClass` will be generated by sending the message `new` to the class, i.e.: $\texttt{mem} \triangleq \texttt{memClass.new} : Mem$. More generally, if a class instance can be typed with $Type \triangleq \texttt{obj}\,t.[\texttt{m}_i\upsilon_i : \sigma_i\{t\}\diamond]^{i\in I}$ then the type of the class whose instances can be typed with $Type$ is $Class(Type) \triangleq [\texttt{new} : Type, \texttt{pre-m}_i :$ $\forall(u <: Type)u{\to}\sigma_i\{u\}]^{i\in I}$. As an interesting remark, we note that the type of class instances is a diamond-type: as such, all class instances can be dynamically extended by new methods (in pure prototype-based style).

Modelling Inheritance. Given an object-type $Type'$ (we consider a diamond-type, but we can consider a saturated-type as well) of the shape $\texttt{obj}\,t.[\texttt{m}_i\upsilon_i : \tau_i\{t\}\diamond]^{i\in I\cup J}$ and a class type $Class(Type') \triangleq [\texttt{new}{:}Type', \texttt{pre-m}_i : \forall(u <: Type')u{\to}\tau_i\{u\}]^{i\in I\cup J}$ we can say that forall $i \in I$, a pre-method $\texttt{pre-m}_i$ is inheritable from $Class(Type)$ to $Class(Type')$ if and only if $u <: Type'$ implies $\sigma_i\{u\} <: \tau_i\{u\}$. As in [AC95], the above condition hold for invariant and contravariant components, but not necessarily for covariant components. We overcome this restriction on covariant components using object extension. A detailed treatment of inheritance can be found in [AC95].

5 Related Work

This section is devoted to a comparison between some interesting and related works appeared in the literature in the last few years.
[Rém98] A calculus very close to $\mathcal{O}b^{+}_{s<:}$ is the one of Didier Rémy. In this calculus, objects have the shape $\zeta(\chi,\tau)[\texttt{m}_i = \varsigma(s_i)o_i]^{i\in I}$, where ζ is a binder for types, τ denotes the type of the whole object, i.e `selftype`, χ is a type-variable that also denotes `selftype` (being that in the type rules $s_i{:}\chi$), $\texttt{m}_i$ are the methods contained in the object with relative bodies $\varsigma(s_i)o_i$.

[1] If one want to play with $\mathcal{O}b^{+}_{s<:}$, one may add polymorphic types and type abstraction/application, following Section 4 of [AC95].

Let $\text{o} \triangleq \zeta(\chi,\tau)[\mathtt{m}_i = \varsigma(s_i)\text{o}_i]^{i\in I}$. Also in the calculus of Rémy, it is possible to extend objects with new methods; when we extend an object with a method m (in our notation $\text{o.m} := \varsigma(s{:}u <: \tau)\text{o}'$) this reduces to $\zeta(\chi, \tau \leftarrow \tau')[\mathtt{m}_i{=}\varsigma(s_i)\text{o}_i, \mathtt{m}{=}\varsigma(s)\text{o}]^{i\in I}$, where τ' is the type of `self` in the body of m, and $\tau \leftarrow \tau$' is the new type of `self` obtained by suitable type reduction rules, necessary to maintains programs both well-formed and well-typed (but the operational semantics is still not type-driven). While there are similarities with our proposal and the one of [Rém98] - notably the use of subtyping for dealing with object extension - the two calculi have some fundamental differences:

- in [Rém98] after an object update, the type of `self` must be "recompiled" using the $\leftarrow$ function, since the type of `self` is factorised by all methods; this is not the case in $\mathcal{O}b^{+}_{s<:}$ because of a "redundancy" of type annotations inside method bodies;
- the [Rém98] calculus have the, so called, `virtual` methods (absent in $\mathcal{O}b^{+}_{s<:}$);
- the $\mathcal{O}b^{+}_{s<:}$ calculus have override of explicit annotations (absent in [Rém98]);
- variance annotations are the same in both calculi, but private-invariant annotation is absent in [Rém98].
- in $\mathcal{O}b^{+}_{s<:}$ we distinguish between two shape of objects, namely extendible objects, and "fixed-size" objects, while in [Rém98] all object are taken to be extendible;
- in [Rém98], object-types are interpreted as total functions from method labels to types, while in $\mathcal{O}b^{+}_{s<:}$ we rely on the more conventional interpretation of object-types as partial functions.

[RS98] The paper of Riecke and Stone describes a functional Object Calculus à la Abadi and Cardelli that allows unrestricted object extension in presence of object subsumption. The novelty of this paper is that we can forget a method with type σ and later re-add it with a type τ *incompatible* with σ. This can be done by distinguish "external" method names by "internal" ones. A proper "dictionary" is attached to each object in order to"link" external labels to internal labels. Private fields can be hidden from the outside by subsumption.

One of the novelty of this paper is the operational semantics that at each step manipulates method dictionaries. This manipulation has a run-time cost that can slowly the running of the program, although some optimization techniques are proposed by the authors. Moreover the style of programming induced by adding dictionaries has an impact on the style of programming, since after a while of extensions and subsumptions steps one must reconstruct the correct behaviour of some methods.

[AC95] This paper is the "father" of the present paper; many of the ideas present in this paper have stimulated our development. The *Imperative Object Calculus* is to our knowledge the first object calculus with an imperative semantics, a sound type system with `selftypes`, subtyping and variance annotations.

References

[Aba94] M. Abadi. Baby Modula–3 and a Theory of Objects. *Journal of Functional Programming*, 4(2):249–283, 1994.

[AC95] M. Abadi and L. Cardelli. An Imperative Object Calculus. In *Proc. of TAP-SOFT/FASE*, Lecture Notes in Computer Science, pages 471–485. Springer-Verlag, 1995. Also in *Theory and Practice of Object Systems* 1(3):151-166, 1995.

[AC96a] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[AC96b] M. Abadi and L. Cardelli. A Theory of Primitive Objects: Untyped and First Order Systems. *Information and Computation*, 125(2):78–102, 1996.

[BBDL97] 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.

[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.

[BL95] 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.

[BSvG95] K.B. Bruce, A. Shuett, and R. van Gent. Polytoil: a Type-safe Polymorphic Object Oriented Language. In *Proc. of ECOOP*, volume 952 of *Lecture Notes in Computer Science*, pages 16–30. Springer-Verlag, 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.

[CU89] C. Chambers and D. Ungar. Customization: Optimizing Compiler Technology for Self, a Dinamically-typed Object-Oriented Programming Language. In *SIGPLAN-89 Conference on Programming Language Design and Implementation*, pages 146–160, 1989.

[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 of *Lecture Notes in Computer Science*, pages 844–885. Springer-Verlag, 1994.

[FM95] 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.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: the Language and its Implementation*. Addison-Wesley, 1983.

[Liq97a] L. Liquori. Bounded Polymorphism for Extensible Objects. Technical Report CS-24-96, Computer Science Department, University of Turin, Italy, 1997.

[Liq97b] 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.

[Liq98] L. Liquori. On Object Extension. In *Proc. of ECOOP*, volume 1445 of *Lecture Notes in Computer Science*, pages 498–552. Springer-Verlag, 1998.

[Liq99] L. Liquori. Bounded Polymorphism for Extensible Objects. Technical Report RR 1999-16, École Normale Supérieure de Lyon, France, 1999.

[Mey92] B. Meyer. *Eiffel:The language*. Prentice Hall, 1992.

[Mic90] J. C. Michell. Toward a Typed Foundation for Method Specialization and Inheritance. In *Proceedings of POPL*, pages 109–124. The ACM Press, 1990.

[Rém95] D. Rémy. Refined Subtyping and Row Variables for Record Types. Draft, 1995.

[Rém98] D. Rémy. From Classes to Objects via Subtyping. In *Proc. of ESOP*, volume 1381 of *Lecture Notes in Computer Science*, pages 200–220. Springer-Verlag, 1998.

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

[US87] D. Ungar and B. Smith, R. Self: The Power of Simplicity. In *Proc. of OOPSLA*, pages 227–241. The ACM Press, 1987.

A The Extended Object Calculus

Well-formed Contexts

$$\frac{}{\varepsilon \vdash ok}\ (C{-}\varepsilon) \qquad \frac{\Gamma \vdash \sigma \qquad s \notin dom(\Gamma)}{\Gamma, s : \sigma \vdash ok}\ (C{-}s) \qquad \frac{\Gamma \vdash \sigma \qquad t \notin dom(\Gamma)}{\Gamma, t <: \sigma \vdash ok}\ (C{-}t)$$

Well-formed Types

$$\frac{\begin{array}{l}\Gamma, t <: \omega \vdash \sigma_i\{t^+\} \quad \forall\, i \in I \quad I \cap J = \emptyset \\ \Gamma, t <: \omega \vdash \sigma_j\{t^+\} \quad \forall j \in J \quad \upsilon_j \in \{^\circ, ^-\}\end{array}}{\Gamma \vdash \mathtt{obj}\, t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\} \diamond \mathtt{m}_j \upsilon_j : \sigma_j\{t\}]^{i \in I}_{j \in J}}\ (T{-}\diamond) \qquad \frac{\Gamma, t <: \sigma, \Gamma' \vdash ok}{\Gamma, t <: \sigma, \Gamma' \vdash t}\ (T{-}Var)$$

$$\frac{\Gamma, t <: \omega \vdash \sigma_i\{t^+\} \qquad \forall\, i \in I}{\Gamma \vdash \mathtt{obj}\, t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\}]^{\,i \in I}}\ (T{-}Sat) \qquad \frac{\Gamma \vdash ok}{\Gamma \vdash \omega}\ (T{-}\Omega)$$

Subtyping Judgments with Variance Annotations

$$\frac{\Gamma \vdash \sigma <: \sigma' \qquad \upsilon \in \{^\circ, ^+\}}{\Gamma \vdash \upsilon\sigma <:{}^{+}\sigma'}\ (S{-}Cova) \qquad \frac{\Gamma \vdash \sigma' <: \sigma \qquad \upsilon \in \{^\circ, ^-\}}{\Gamma \vdash \upsilon\sigma <:{}^{-}\sigma'}\ (S{-}Contra)$$

$$\frac{\Gamma \vdash \sigma \qquad \upsilon \in \{^\circ, ^\bullet\}}{\Gamma \vdash \upsilon\sigma <: \upsilon\sigma}\ (S{-}Inv_1) \qquad \frac{\Gamma \vdash \sigma \qquad \upsilon \in \{^+, ^-\}}{\Gamma \vdash \upsilon\sigma <:{}^{\bullet}\sigma}\ (S{-}Inv_2)$$

Standard Subtyping Judgments

$$\frac{\Gamma \vdash \sigma}{\Gamma \vdash \sigma <: \sigma}\ (S{-}Refl) \qquad \frac{\Gamma \vdash \sigma <: \tau \qquad \Gamma \vdash \tau <: \rho}{\Gamma \vdash \sigma <: \rho}\ (S{-}Trans) \qquad \frac{\Gamma \vdash \sigma}{\Gamma \vdash \sigma <: \omega}\ (S{-}\Omega)$$

Subtyping Judgments for Object-Types

$(S{-}Var_\diamond)$

$$\frac{\Gamma, u <: \mathtt{obj}\, t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\} \diamond \mathtt{m}_j \upsilon_j : \sigma_j\{t\}]^{i \in I}_{j \in J} \vdash \upsilon_k \sigma_k\{u\} <: \upsilon'_k \sigma'_k\{u\} \qquad \forall\, k \in I \cup J}{\Gamma \vdash \mathtt{obj}\, t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\} \diamond \mathtt{m}_j \upsilon_j : \sigma_j\{t\}]^{i \in I}_{j \in J} <: \mathtt{obj}\, t.[\mathtt{m}_i \upsilon'_i : \sigma'_i\{t\} \diamond \mathtt{m}_j \upsilon'_j : \sigma'_j\{t\}]^{i \in I}_{j \in J}}$$

$$\frac{\Gamma, u <: \mathtt{obj}\, t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\}]^{\,i \in I} \vdash \upsilon_k \sigma_k\{u\} <: \upsilon'_k \sigma'_k\{u\} \qquad \forall\, k \in I}{\Gamma \vdash \mathtt{obj}\, t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\}]^{\,i \in I} <: \mathtt{obj}\, t.[\mathtt{m}_i \upsilon'_i : \sigma'_i\{t\}]^{\,i \in I}}\ (S{-}Var)$$

$(S{-}Shift_\diamond)$

$$\frac{\Gamma \vdash \mathtt{obj}\, t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\} \diamond \mathtt{m}_j \upsilon_j : \sigma_j\{t\}]^{i \in I \cup K}_{j \in J} \qquad \upsilon_k \in \{^\circ, ^-\} \qquad \forall\, k \in K}{\Gamma \vdash \mathtt{obj}\, t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\} \diamond \mathtt{m}_j \upsilon_j : \sigma_j\{t\}]^{i \in I \cup K}_{j \in J} <: \mathtt{obj}\, t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\} \diamond \mathtt{m}_j \upsilon_j : \sigma_j\{t\}]^{i \in I}_{j \in J \cup K}}$$

$(S{-}Ext_\diamond)$

$$\frac{\Gamma \vdash \mathtt{obj}\, t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\} \diamond \mathtt{m}_j \upsilon_j : \sigma_j\{t\}]^{i \in I}_{j \in J \cup K} \qquad \upsilon_k \in \{^\circ, ^-\} \qquad \forall\, k \in K}{\Gamma \vdash \mathtt{obj}\, t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\} \diamond \mathtt{m}_j \upsilon_j : \sigma_j\{t\}]^{i \in I}_{j \in J} <: \mathtt{obj}\, t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\} \diamond \mathtt{m}_j \upsilon_j : \sigma_j\{t\}]^{i \in I}_{j \in J \cup K}}$$

$$\frac{\varGamma \vdash \mathtt{obj}\,t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\} \diamond \mathtt{m}_j \upsilon_j : \sigma_j\{t\}]_{j\in J}^{i\in I}}{\varGamma \vdash \mathtt{obj}\,t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\} \diamond \mathtt{m}_j \upsilon_j : \sigma_j\{t\}]_{j\in J}^{i\in I} <: \mathtt{obj}\,t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\}]^{i\in I}} \quad (S{-}Sat_\diamond)$$

$$\frac{\varGamma \vdash \mathtt{obj}\,t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\}]^{i\in I\cup J}}{\varGamma \vdash \mathtt{obj}\,t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\}]^{i\in I\cup J} <: \mathtt{obj}\,t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\}]^{i\in I}} \quad (S{-}Width)$$

Type Rules for Objects

$$\frac{\varGamma, s : \sigma, \varGamma' \vdash ok}{\varGamma, s : \sigma, \varGamma' \vdash s : \sigma} \quad (V{-}Proj) \qquad \frac{\varGamma \vdash \mathtt{o} : \sigma \qquad \varGamma \vdash \sigma <: \tau}{\varGamma \vdash \mathtt{o} : \tau} \quad (V{-}Sub)$$

$$\frac{\varGamma \vdash \mathtt{o} : \tau \qquad \varGamma \vdash \tau <: \mathtt{obj}\,t.[\mathtt{m}_k \upsilon_k : \sigma_k\{t\}] \qquad \upsilon_k \in \{^\circ, ^+\}}{\varGamma \vdash \mathtt{o.m}_k : \sigma_k\{\tau\}} \quad (V{-}Sel)$$

(Let $\tau_i \triangleq \mathtt{obj}\,t.[\mathtt{m}_h \upsilon_h : \sigma_h\{t\}]^{h\in H_i\cup\{i\}}$).

$$\frac{\varGamma, s_i : u <: \tau_i \vdash \mathtt{o}_i : \sigma_i\{u\} \qquad H_i \subseteq I \qquad \varUpsilon_i : \upsilon_i \qquad \forall i \in I}{\varGamma \vdash [\mathtt{m}_i \varUpsilon_i = \varsigma(s_i{:}u <: \tau_i)\mathtt{o}_i]^{i\in I} : \mathtt{obj}\,t.[\mathtt{m}_i \upsilon_i : \sigma_i \diamond]^{i\in I}} \quad (V{-}Obj)$$

$$\frac{\begin{array}{l}\varGamma \vdash \mathtt{o} : \tau \qquad \varGamma \vdash \tau <: \mathtt{obj}\,t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\}]^{i\in I} \qquad k \in I\\ \varGamma, s_k : u <: \mathtt{obj}\,t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\}]^{i\in I} \vdash \mathtt{o}' : \sigma_k\{u\} \qquad \upsilon_k \in \{^\circ, ^-\}\end{array}}{\varGamma \vdash \mathtt{o.m}_k := \varsigma(s_k{:}u <: \mathtt{obj}\,t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\}]^{i\in I})\mathtt{o}' : \tau} \quad (V{-}Over)$$

$$\frac{\begin{array}{l}\varGamma \vdash \mathtt{o} : \mathtt{obj}\,t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\}]^{i\in I}\\ \varGamma \vdash \mathtt{obj}\,t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\}]^{i\in I} <: \mathtt{obj}\,t.[\mathtt{m}_k \upsilon : \sigma_k\{t\}] \qquad \varUpsilon : \upsilon\end{array}}{\varGamma \vdash \mathtt{o.m}_k := \varUpsilon : \mathtt{obj}\,t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\}, \mathtt{m}_k \upsilon : \sigma_k\{t\}]^{i\in I\setminus\{k\}}} \quad (V{-}Ann_1)$$

$$\frac{\begin{array}{l}\varGamma \vdash \mathtt{o} : \mathtt{obj}\,t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\} \diamond \mathtt{m}_j \upsilon_j : \sigma_j\{t\}]_{j\in J}^{i\in I}\\ \varGamma \vdash \mathtt{obj}\,t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\} \diamond \mathtt{m}_j \upsilon_j : \sigma_j\{t\}]_{j\in J}^{i\in I} <: \mathtt{obj}\,t.[\mathtt{m}_k \upsilon : \sigma_k\{t\}] \qquad \varUpsilon : \upsilon\end{array}}{\varGamma \vdash \mathtt{o.m}_k := \varUpsilon : \mathtt{obj}\,t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\}, \mathtt{m}_k \upsilon : \sigma_k\{t\} \diamond \mathtt{m}_j \upsilon_j : \sigma_j\{t\}]_{j\in J}^{i\in I\setminus\{k\}}} \quad (V{-}Ann_2)$$

(Let $\tau_k \triangleq \mathtt{obj}\,t.[\mathtt{m}_h \upsilon_h : \sigma_h\{t\}]^{h\in H\cup\{k\}}$).

$$\frac{\begin{array}{ll}\varGamma \vdash \mathtt{o} : \mathtt{obj}\,t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\} \diamond \mathtt{m}_j \upsilon_j : \sigma_j\{t\}]_{j\in J}^{i\in I} & k \in J\\ \varGamma, s_k : u <: \tau_k \vdash \mathtt{o}' : \sigma_k\{u\} \qquad \varUpsilon : \upsilon_k & H \subseteq I\end{array}}{\varGamma \vdash \mathtt{o.m}_k := \varUpsilon\varsigma(s_k{:}u <: \tau_k)\mathtt{o}' : \mathtt{obj}\,t.[\mathtt{m}_i \upsilon_i : \sigma_i\{t\} \diamond \mathtt{m}_j \upsilon_j : \sigma_j\{t\}]_{j\in J\setminus\{k\}}^{i\in I\cup\{k\}}} \quad (V{-}Ext)$$