From Classes to Objects via Subtyping*

Didier Rémy

INRIA-Rocquencourt**

Abstract

We extend the Abadi-Cardelli calculus of primitive objects with object extension. We enrich object types with a more precise, uniform, and flexible type structure. This enables to type object extension under both width and depth subtyping. Objects may also have extend-only or virtual contra-variant methods and read-only co-variant methods. The resulting subtyping relation is richer, and types of objects can be weaken progressively from a class level to a more traditional object level along the subtype relationship.

1 Introduction

Object extension has long been considered unsound when combined with subtyping. The problem may be explained as follows: in an object built with two methods ℓ_1 and ℓ_2 of types τ_1 and τ_2, the method ℓ_1 may require ℓ_2 to be of type τ_2. Forgetting the method ℓ_2 by subtyping would result in the possible redefinition of method ℓ_2 with another, incompatible type τ_3. Then, the invocation of ℓ_1 may fail.

Indeed, the first strongly-typed object-based languages that have been proposed provided either subtyping [1] or object extension [21] to circumvent the problem described above. However, each proposal was missing an important feature supported by the other one.

Both of them were improved later following the same principle: At an earlier stage, object components were assembled in prototypes [20] or classes [2], relying on some extension mechanism to provide inheritance. Objects were formed in a second, atomic step, immediately losing their extension capabilities for ever, to the benefit of subtyping.

In contrast to the previous work, we allow both extension and subtyping at the level of objects, avoiding stratification. Our solution is based on the enrichment of the structure of object types. Thus, our type-system rejects the above counter-example while keeping many other useful programs. In our proposal, an object and its class are unified and can be considered as two different perspectives on the same value: the type of an object is a supertype of the type of its class. Fine grain subtyping allows type information to be lost gradually, both width-wise and depth-wise, slowly fading classes into objects. As is well-known, when more type information is exposed, more operations can be performed (class perspective). On the contrary, hiding a sufficient amount of type information allows for more object interchangeability, but permits fewer operations (object perspective).

We add object extension to the object calculus of Abadi and Cardelli [3]. We adapt their typing rules to our enriched object types. In particular, we force methods to be parametric in self, that is, polymorphic over all possible extensions of the respective object. In this sense, our proposal is not a strict extension of theirs.

In addition to object extension, the enriched type structure has other benefits. We can allow virtual methods in objects (*i.e.* methods that are required by some other method but that have

*A preliminary version appeared in [26]

**BP 105, 78153 Le Chesnay Cedex, France. Email: `Didier.Remy@inria.fr`

not been defined yet) since we are able to described them in types. Using co-variant subtyping forbids further re-definition of the corresponding method, as in [3]. Since classes are objects, such methods are in fact final methods. Final methods can only be accessed but no more redefined (except, indirectly, by the invocation of a previously defined method).

Virtual methods are useful because they allow objects to be built progressively, component by component, rather than all at once. They also improve security, since they sometime avoid the artificial use of dangerous default methods. While final methods are co-variant, virtual methods, are naturally contra-variant.

The rest of the paper is organized as follows. In the next section, we describe our solution informally. The following section is dedicated to the formal presentation. In section 4, we show some properties of the type system, in particular the type soundness property. Section 5 illustrates the gain in security and flexibility of our proposal by running a few examples. To a large extend, these examples can be understood intuitively and may also be read simultaneously with or immediately after the informal presentation. In section 6 we discuss possible extensions and variations of our proposal, as well as further meta-theoretical developments. A brief comparison with other works is done in section 7 before concluding.

2 Informal presentation

Technically, our first goal is to provide method extension, while preserving some form of subtyping. The counter-example given above does not imply that both method extension and width subtyping are in contradiction. It only shows that combining two existing typing rules would allow to write unsafe programs. Thus, if ever possible, a type system with both method extension and subtyping should clearly impose restrictions when combining them. Our solution is to enrich types so that subtyping becomes traceable, and so that extension can be limited to those fields whose exact type is known.

We first recall record types with symmetric type information. Using a similar structure for object types, some safe uses of subtyping and object extension can be typed, while the counter-example given in the introduction is rejected.

Record types

Record values are partial functions with finite domains that map labels to values. Traditionally, the types of records are also partial functions with finite domains that map labels to types. They are represented as records of types, that is, $\{\ell_i : \tau_i{}^{i \in I}\}$. This type says that fields ℓ_i's are defined with values of type τ_i's. However, it does not imply anything about other fields.

Another richer, more symmetric structure has also been used for record types, originally to allow type inference for records in ML [23, 24]. There, record types are treated as total functions mapping labels to field types, with the restriction that all but a finite number of labels have isomorphic images (*i.e.* are equal modulo renaming). Thus, record types can still be represented finitely by listing all significant labels with their corresponding field types and then adding an extra field-type acting as a template for all other labels.

In their simplest form, field types are either $\mathtt{P}\ \tau$ (read *present* with type τ) or $\mathtt{A}$ (read *absent*). For instance, a record with two fields ℓ_1 of type τ_1 and ℓ_2 of type τ_2 is given type $\langle \ell_1 \colon \mathtt{P}\ \tau_1 \ ; \ell_2 \colon \mathtt{P}\ \tau_2 \ ; \mathtt{A} \rangle$. It could also, equivalently, be given type $\langle \ell_1 \colon \mathtt{P}\ \tau_1 \ ; \ell_2 \colon \mathtt{P}\ \tau_2 \ ; \ell \colon \mathtt{A} \ ; \mathtt{A} \rangle$ where ℓ is distinct from ℓ_1 and ℓ_2.

In the absence of subtyping, standard types for records $\{\ell_i : {\tau_i}^{i \in I}\}$ can indeed be seen as a special case of record types, where field variables are disallowed; their standard subtyping relation then corresponds to the one generated by the axiom $\mathtt{P}\ \tau <: \mathtt{A}$ (and obvious structural rules). The type $\{\ell_1 : \tau_1; ..\ell_n : \tau_n\}$ becomes an abbreviation for $\langle \ell_1 : \mathtt{P}\ \tau_1\ ;\ ..\ell_n : \mathtt{P}\ \tau_n\ ;\ \mathtt{A}\rangle$. However, record types are much more flexible. For instance, they inherently and symmetrically express negative information. Before we added subtyping, a field ℓ of type $\mathtt{A}$ was known to be absent in the corresponding record. This is quite different from the absence of information about field ℓ. Such precise information is sometimes essential; a well-known example is record concatenation [16]. Instead of breaking the symmetry with the subtyping axiom $\mathtt{P}\ \tau <: \mathtt{A}$, we might have introduced a new field $\mathtt{U}$ (read *unknown*), with two axioms $\mathtt{P}\ \tau <: \mathtt{U}$ and $\mathtt{A} <: \mathtt{U}$. This would preserve the property that a field of type $\mathtt{A}$ is known to be absent, still allowing present and absent field to be interchanged but at their common supertype $\mathtt{U}$.

Field variables and row variables also increase the expressiveness of record types. However, for simplicity, we do not take this direction here. Below, we use meta-variables for rows. This is just a notational convenience. It does not add any power.

Object types

In their simplest form, objects are just records, thus object types mimic record types. We write object types with $[\rho]$ instead of $\langle\rho\rangle$ to avoid confusion. An object with type $[\ell_1 : \mathtt{P}\ \tau_1\ ;\ \ell_2 : \mathtt{P}\ \tau_2\ ;\ \mathtt{A}]$ possesses two methods ℓ_1 and ℓ_2 of respective types τ_1 and τ_2. Intuitively, an object $[\ell_1 = {a_i}^{i \in I}]$ can be given type $[\ell_i : \mathtt{P}\ {\tau_i}^{i \in I}\ ;\ \mathtt{A}]$ provided methods a_i's have type τ_i's.

However, objects soon differ from records by their ability to send messages to themselves, or to return themselves in response to a method call. More generally, objects are of the form $[\ell_i = \varsigma(x_i)a_i]$. Here, x_i is a variable that is bound to the object itself when the method ℓ_i is invoked. Consistently, the expression a_i must be typed in a context where x_i is assumed of the so-called "mytype", represented by some type variable χ equal to the object type τ. The following typing rule is a variant of the one used in [3].

$$\frac{\tau \equiv \zeta(\chi)[\ell_i : \mathtt{P}\ {\tau_i}^{i \in I}\ ;\ \mathtt{A}] \qquad A, \chi = \tau, x_i : \chi \vdash a_i : \tau_i}{A \vdash \zeta(\chi, \tau)[\ell_i = \varsigma(x_i){a_i}^{i \in I}] : \tau}$$

(The type annotation (χ, τ) in the object expression binds the name of mytype locally and specifies the type of the object.)

An extendible object v may also be used to build a new object v' with more methods than v and thus of a different type, say τ'. The type τ' of self in v' is different from the type τ of self in v. In order to remain well-typed in v', the methods of v, should have been typed in a context where the type of self could have been τ' as well as τ. This applies to any possible extension v' of v. In other words, methods of an object of type τ should be parametric in all possible types of all possible successive extensions of an object of type τ. This condition can actually be expressed with subtyping by $\chi <: \#\,\tau$, where $\#\,\tau$ is called the *extension type* of τ (also called the internal type of the object). That is, the least upper bound of all exact[1] types of complete extensions (extensions in which no virtual method remains) of objects of external type τ.

A field of type $\mathtt{A}$ can be overridden with methods of arbitrary types. Thus, the best type for that field in the self parameter is $\mathtt{U}$, *i.e.* we choose $\#\mathtt{A}$ to be $\mathtt{U}$. Symmetrically, we choose $\#(\mathtt{P}\ \tau)$ to be $\mathtt{U}$. This makes methods of type $\mathtt{P}\ \tau$ internally unaccessible. Fields of type $\mathtt{P}\ \tau$ are known

[1]The exact type of an object is the type with which the methods can initially be typed. The external type of an object may be a supertype of the exact type.

to be present externally, but are not assumed to be so internally. Thus, fields of type $\mathtt{P}\ \tau$ can be overridden with methods of arbitrary types, such as fields of type $\mathtt{A}$. To recover the ability to send messages to self, we introduce a new type field $\mathtt{R}\ \tau$ (read *required* of type τ). A field of type $\mathtt{R}\ \tau$ is defined with a method of type τ, and is required to remain of at least type τ, internally. Such a field can only be overridden with a method of type τ. Therefore, self can also view it as a field that is, and will remain, of type τ. In math, $\#\mathtt{R}\ \tau$ is $\mathtt{R}\ \tau$. A field of type $\mathtt{P}\ \tau$, can safely be considered as a field of type $\mathtt{R}\ \tau$. Thus, we assume $\mathtt{P}\ \tau <: \mathtt{R}\ \tau$. We also assume $\mathtt{R}\ \tau$ to be a subtype of $\mathtt{U}$. As an example, $\#\,\zeta(\chi)[\ell_1{:}\,\mathtt{R}\ \tau_1\ ;\ \ell_2{:}\,\mathtt{P}\ \tau_2\ ;\ \ell_3{:}\,\mathtt{U}\ ;\ \mathtt{A}]$ is $\zeta(\chi)[\ell_1{:}\,\mathtt{R}\ \tau_1\ ;\ \ell_2{:}\,\mathtt{U}\ ;\ \ell_3{:}\,\mathtt{U}\ ;\ \mathtt{U}]$, or shortly $\zeta(\chi)[\ell_1{:}\,\mathtt{R}\ \tau\ ;\ \mathtt{U}]$.

The extension of a field with a method of type τ requires that field to be either of type $\mathtt{A}$ or $\mathtt{R}\ \tau$ in the original record (the field may also be of type $\mathtt{P}\ \tau$, which is a subtype of $\mathtt{R}\ \tau$.) It is possible to factor the two cases by introducing a new field type $\mathtt{M}\ \tau$ (read *maybe* of type τ), and the axioms $\mathtt{R}\ \tau <: \mathtt{M}\ \tau$, $\mathtt{A} <: \mathtt{M}\ \tau$, and $\mathtt{M}\ \tau <: \mathtt{U}$. Intuitively, $\mathtt{M}\ \tau$ is the union type $\mathtt{R}\ \tau \cup \mathtt{A}$. This allows, in a first step, to ignore the presence of a method while retaining its type, and, in a second step, to forget the type itself. The type of object extension becomes more uniform. Roughly, if the original object has type $[\ell_1{:}\,\mathtt{M}\ \tau_1\ ;\ \tau_2]$ and the new method ℓ_1 has type τ_1 then the resulting object has type $[\ell_1{:}\,\mathtt{R}\ \tau_1\ ;\ \tau_2]$.

A field of type $\mathtt{M}\ \tau$ may later be defined or redefined with some method of type τ, becoming of type $\mathtt{R}\ \tau$, which is a subtype of $\mathtt{M}\ \tau$. It may also be left unchanged and thus remain of type $\mathtt{M}\ \tau$. Thus, a field of type $\mathtt{M}\ \tau$ will always remain of a subtype of $\mathtt{M}\ \tau$. That is, $\#(\mathtt{M}\ \tau)$ is $\mathtt{M}\ \tau$.

Deep subtyping

Subtyping rules described so far allow for width subtyping but not for depth subtyping, since all constructors have been left invariant. The only constructor that could be made covariant without breaking type-soundness is $\mathtt{P}$. Making $\mathtt{R}$ co-variant would be unsafe. However, we can safely introduce a new field type $\mathtt{R}^+\ \tau$ to tell that a method is defined and required to be of a subtype of τ, provided that a field of type $\mathtt{R}^+\ \tau$ is never overridden. On the other hand, a method ℓ_1 can safely be invoked on any object of type $[\ell_1{:}\,\mathtt{R}^+\ \tau_1\ ;\ \mathtt{U}]$, which returns an expression of type τ_1. Of course, we also add $\mathtt{R}\ \tau <: \mathtt{R}^+\ \tau$ to just forget the fact that we are revealing the exact type information.

Symmetrically, a field ℓ of type $\mathtt{M}\ \tau$ cannot be accessed, but it can be redefined with a method of a subtype of τ. Still, it would be unsound to make $\mathtt{M}\ \tau$ contra-variant. By contradiction, consider an object p of type $\zeta(\chi)[\ell{:}\,\mathtt{R}\ \tau\ ;\ \ell'{:}\,\mathtt{P}\ \chi\ ;\ \mathtt{A}]$ where calling method ℓ' overrides ℓ in self with a new method of type τ. By subtyping p_0 could be given type $\zeta(\chi)[\ell{:}\,\mathtt{M}\ \tau_0\ ;\ \ell'{:}\,\mathtt{P}\ \chi\ ;\ \mathtt{A}]$ where τ_0 is a subtype of τ. Then let p_2 of type $\zeta(\chi)[\ell{:}\,\mathtt{M}\ \tau_0\ ;\ \ell'{:}\,\mathtt{P}\ \chi\ ;\ \ell''{:}\,\mathtt{P}\ \mathtt{unit}\ ;\ \mathtt{A}]$ be the extension of p_1 with a new method ℓ'' that requires ℓ of type τ_0. Calling method ℓ' of p_2 restore field ℓ of p_2 to some method of type τ and returns an object p_3. However, calling method ℓ'' of p_3 expects a method ℓ of type τ_0 but finds one of type τ.

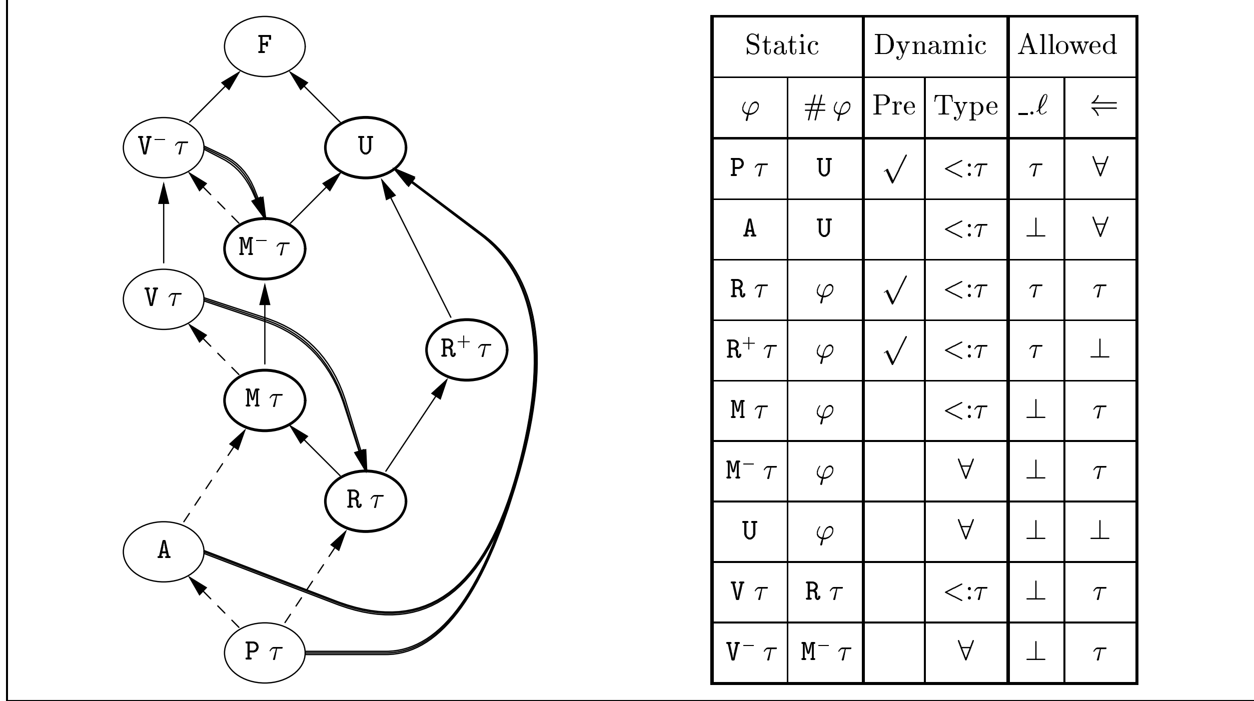
We can still introduce a contra-variant symbol $\mathtt{M}^-$ with the axiom $\mathtt{M}\ \tau <: \mathtt{M}^-\ \tau$. Then, a method $\mathtt{M}^-\ \tau$ can be redefined, but the method in the resulting object remains of type $\mathtt{M}^-\ \tau$ and is thus unaccessible. This is still useful in situations where contra-variance is mandatory or to enforce protection against accidental access (see sections 5.6, 5.2 and [3].)

Virtual methods

A method ℓ is *virtual* with type τ (which we write $\mathtt{V}\ \tau$) if other methods have assumed ℓ to be of type $\mathtt{R}\ \tau$, while the method itself might not have been defined yet. When an object has a virtual method, no other method of that object can be invoked. Thus, $\mathtt{V}\ \tau$ should not be a subtype of $\mathtt{U}$. A method

of type $\mathtt{V}\ \tau$ can be extended as a method of type $\mathtt{R}\ \tau$. Virtual methods may also be contra-variant. We use another symbol $\mathtt{V}^-\ \tau$ to indicate that deep subtyping has been used. A contra-variant virtual method can be extended, but it must remain contra-variant after its extension, *i.e.* of type $\mathtt{M}^-\ \tau$, and thus inaccessible. This may be surprising at first. The intuition if that $\#(\mathtt{V}^-\ \tau)$ should be $\mathtt{R}^-\ \tau$. However, a method of field-type $\mathtt{R}^-\ \tau$ would be inaccessible, since its best type is unknown. Thus $\mathtt{R}^-\ \tau$ has been identified with $\mathtt{M}^-\ \tau$.

For convenience, we also introduce a new constant $\mathtt{F}$ that is a top type for fields. That is, we assume $\mathtt{V}^-\ \tau <: \mathtt{F}$ and $\mathtt{U} <: \mathtt{F}$ (all other relations hold by transitivity).



Static		Dynamic		Allowed	
φ	$\#\varphi$	Pre	Type	$_.\ell$	$\Leftarrow$
$\mathtt{P}\ \tau$	$\mathtt{U}$	$\surd$	$<:\tau$	τ	$\forall$
$\mathtt{A}$	$\mathtt{U}$		$<:\tau$	$\perp$	$\forall$
$\mathtt{R}\ \tau$	φ	$\surd$	$<:\tau$	τ	τ
$\mathtt{R}^+\ \tau$	φ	$\surd$	$<:\tau$	τ	$\perp$
$\mathtt{M}\ \tau$	φ		$<:\tau$	$\perp$	τ
$\mathtt{M}^-\ \tau$	φ		$\forall$	$\perp$	τ
$\mathtt{U}$	φ		$\forall$	$\perp$	$\perp$
$\mathtt{V}\ \tau$	$\mathtt{R}\ \tau$		$<:\tau$	$\perp$	τ
$\mathtt{V}^-\ \tau$	$\mathtt{M}^-\ \tau$		$\forall$	$\perp$	τ

Figure 1: Structure of field types

The final structure of field types and subtyping axioms are summarized in figure 1. Thick arrows represent the function $\#$. Thick nodes are used instead of reflexive thick arrows, that is, thick nodes are left invariant by $\#$. Thin arrows represent subtyping. We added a redundant but useful distinction between continuous and dashed thin arrows. They are respectively covariant and contra-variant by type-extension: when a continuous arrow connects τ_1 and τ_2, then $\#\,\tau_1$ is also a subtype of $\#\,\tau_2$; the inverse applies to dashed arrows.

Although it is easy to give intuitions for parts of the hierarchy taken alone (variances, virtual methods, idempotent field-types), we are not able to propose a good intuition for the whole hierarchy. The different components are modular technically, but their intuitive, thus approximative descriptions, cannot be composed here. We think that the field-type hierarchy should be understood locally, and then considered as such.

The table on the right is a summary of field types and their properties. The entry φ in the first column indicates the static external type. The second column $\#\,\varphi$ is its extension type, *i.e.* the static internal type. The two following columns tell whether the field is guaranteed to be present ($\surd$ sign) and its type if present. The reason for having $<:\tau$ instead of τ is the covariance of $\mathtt{P}$. The symbol $\forall$ means any possible type. The last two columns describe access and overriding capabilities

($\bot$ means disallowed).

3 Formal developments

3.1 Types

We assume given a denumerable collection of type variables, written α, β, or χ. Type expressions, written with letter τ, are type variables, object types, or the top type $\mathtt{T}$. An object type $\zeta(\chi)[\ell_i\colon\varphi_i\,^{i\in I}\;;\varphi]$ is composed of a finite sequence of fields $\ell_i : \varphi_i$, without repetition, and a template φ for fields that are not explicitly mentioned. Variable χ is bound in the object type, and should only appear positively in φ_i's as in φ.

$$\begin{aligned}\tau &::= \alpha \mid \zeta(\alpha)[\ell_i\colon\varphi_i\,^{i\in I}\;;\varphi] \mid \mathtt{T}\\ \varphi &::= \mathtt{A} \mid \mathtt{P}\,\tau \mid \mathtt{R}\,\tau \mid \mathtt{M}\,\tau \mid \mathtt{V}\,\tau \mid \mathtt{R}^{+}\,\tau \mid \mathtt{M}^{-}\,\tau \mid \mathtt{V}^{-}\,\tau \mid \mathtt{U} \mid \mathtt{F}\end{aligned}$$

The variance of an occurrence is defined in the usual way: it is the parity of the number of times a variable crosses a contra-variant position (*i.e.*, the number of symbols $\mathtt{V}^{-}$ or $\mathtt{M}^{-}$) on that path from the root to that occurrence. The set of free variables of τ is $fv(\tau)$. We write $fv^{-}(\tau)$ the subset of those variables that occurs negatively at least once.

Object types are considered equal modulo reordering of fields. They are also equal modulo expansion, that is, by extracting a field from the template:

$$\zeta(\chi)[\ell_i\colon\varphi_i\,^{i\in I}\;;\varphi] = \zeta(\chi)[\ell_i\colon\varphi_i\,^{i\in I}\;;\ell\colon\varphi\;;\varphi] \qquad \ell\neq\ell_i, \forall i\in I$$

Rules for the formation of types will be defined jointly with subtyping rules in figure 2 and are described below.

Notation For convenience and brevity of notation, we use meta-variables ρ for rows of fields, that is, syntactic expressions of the form $(\ell_i\colon\varphi_i\,^{i\in I}\;;\varphi_0)$, where φ_i's and I are left implicit. We write $\rho(\ell)$ the value of ρ in ℓ, that is, φ_i if ℓ is one of the ℓ_i's, or φ_0 otherwise. We write $\rho\setminus\ell$ for $(\ell_i\colon\varphi_i\,^{i\in I,\ell_i\neq\ell}\;;\varphi_0)$. and $\ell:\varphi;\rho$ for $(\ell\colon\varphi\;;\ell_i\colon\varphi_i\,^{i\in I,\ell_i\neq\ell}\;;\varphi_0)$. If $\mathcal{R}$ is a relation, we write $\rho\,\mathcal{R}\,\rho'$ for $\forall\ell, \rho(\ell)\,\mathcal{R}\,\rho'(\ell)$.

This is just a meta-notation that is not part of the language of types. It can always be expanded unambiguously into the more explicit notation $(\ell_i\colon\varphi_i\,^{i\in I}\;;\varphi)$.

3.2 Type extension

We define the *extension* of field type φ, written $\#\,\varphi$ by the two first columns of the table 1. Type extension is lifted to object types homomorphically, *i.e.*, $\#\,\zeta(\chi)[\rho]$ is $\zeta(\chi)[\#\,\rho]$. The extension is not defined for type variables, nor for $\mathtt{F}$. Note that the extension is idempotent, that is $\#(\#\,\tau)$ is always equal to $\#\,\tau$.

Well-formation of environments

$$\frac{}{\emptyset \vdash \diamond}\ (\textsc{Env}\ \emptyset) \qquad \frac{E \vdash \tau <: \mathtt{T} \qquad x \notin dom(E)}{E, x : \tau \vdash \diamond}\ (\textsc{Env}\ x) \qquad \frac{E \vdash \tau <: \mathtt{T} \qquad \alpha \notin dom(E)}{E, \alpha <: \tau \vdash \diamond}\ (\textsc{Env}\ \alpha)$$

General subtyping

$$\frac{E, \alpha <: \tau, E' \vdash \diamond}{E, \alpha <: \tau, E' \vdash \alpha <: \tau}\ (\textsc{Sub Var}) \qquad \frac{E \vdash \varphi <: \mathtt{F}}{E \vdash \varphi <: \varphi}\ (\textsc{Sub Ref}\ \mathtt{F}) \qquad \frac{E \vdash \tau <: \mathtt{T}}{E \vdash \tau <: \tau}\ (\textsc{Sub Ref}\ \mathtt{T}) \qquad \frac{E \vdash \tau_1 <: \tau_2 \qquad E \vdash \tau_2 <: \tau_3}{E \vdash \tau_1 <: \tau_3}\ (\textsc{Sub Trans}\ \mathtt{T})$$

$$\frac{E \vdash \varphi_1 <: \varphi_2 \qquad E \vdash \varphi_2 <: \varphi_3}{E \vdash \varphi_1 <: \varphi_3}\ (\textsc{Sub Trans}\ \mathtt{F})$$

Field subtyping (assuming $E \vdash \tau <: \mathtt{T}$)

(Sub PA) $E \vdash \mathtt{P}\,\tau <: \mathtt{A}$ (Sub PR) $E \vdash \mathtt{P}\,\tau <: \mathtt{R}\,\tau$ (Sub AM) $E \vdash \mathtt{A} <: \mathtt{M}\,\tau$ (Sub UF) $E \vdash \mathtt{U} <: \mathtt{F}$ (Sub RR$^+$) $E \vdash \mathtt{R}\,\tau <: \mathtt{R}^+\,\tau$ (Sub RM) $E \vdash \mathtt{R}\,\tau <: \mathtt{M}\,\tau$

(Sub R$^+$U) $E \vdash \mathtt{R}^+\,\tau <: \mathtt{U}$ (Sub MV) $E \vdash \mathtt{M}\,\tau <: \mathtt{V}\,\tau$ (Sub MM$^-$) $E \vdash \mathtt{M}\,\tau <: \mathtt{M}^-\,\tau$ (Sub M$^-$U) $E \vdash \mathtt{M}\,\tau <: \mathtt{U}$ (Sub VV$^-$) $E \vdash \mathtt{V}\,\tau <: \mathtt{V}^-\,\tau$

(Sub V$^-$F) $E \vdash \mathtt{V}^-\,\tau <: \mathtt{F}$

$$\frac{E \vdash \tau <: \tau'}{E \vdash \mathtt{P}\,\tau <: \mathtt{P}\,\tau'}\ (\textsc{Sub PP}) \qquad \frac{E \vdash \tau <: \tau'}{E \vdash \mathtt{R}^+\,\tau <: \mathtt{R}^+\,\tau'}\ (\textsc{Sub R}^+\textsc{R}^+) \qquad \frac{E \vdash \tau <: \tau'}{E \vdash \mathtt{M}^-\,\tau' <: \mathtt{M}^-\,\tau}\ (\textsc{Sub M}^-\textsc{M}^-) \qquad \frac{E \vdash \tau <: \tau'}{E \vdash \mathtt{V}^-\,\tau' <: \mathtt{V}^-\,\tau}\ (\textsc{Sub V}^-\textsc{V}^-)$$

Object subtyping

$$\frac{E \vdash \diamond}{E \vdash \mathtt{T} <: \mathtt{T}}\ (\textsc{Sub TT}) \qquad \frac{E, \chi <: \mathtt{T} \vdash \rho <: \mathtt{F} \qquad \chi \notin fv^-(\rho)}{E \vdash \zeta(\chi)[\rho] <: \mathtt{T}}\ (\textsc{Sub Obj OK})$$

(Sub Obj Invariant) $(\tau \equiv \zeta(\chi)[\rho], \tau' \equiv \zeta(\chi)[\rho'])$

$$\frac{E \vdash \tau <: \mathtt{T} \qquad E \vdash \tau' <: \mathtt{T} \qquad E, \chi <: \mathtt{T} \vdash \rho <: \rho'}{E \vdash \tau <: \tau'}$$

Figure 2: Types and Subtypes

3.3 Expressions

Expressions are variables, objects, method invocation, and method overriding.

$$a ::= x \mid \zeta(\chi, \tau)[\ell_i : \varsigma(x_i) a_i] \mid a.\ell \mid a.\ell \Leftarrow \zeta(\chi, \tau)\varsigma(x)a$$

The expression $a.\ell \Leftarrow \zeta(\chi, \tau)\varsigma(x)a_\ell$ is the extension of a on field ℓ with a method $\varsigma(x)a_\ell$. The expression (χ, τ) binds χ to the type of self in a_ℓ and indicates that the resulting type of the extension should be τ. This information is important so that types do not have to be inferred but

only checked. Field update is just a special case of object extension. This is more general, since the selection between update and extension is resolved dynamically.

3.4 Well formation of types and subtyping

Typing environments are sequences of bindings written with letter E. There are free kinds of judgments (the second and third ones are similar):

$E ::= \emptyset \mid \alpha <: \tau \mid x : \tau$	Typing environments
$E \vdash \diamond$	Environment E is well-formed
$E \vdash \tau <: \tau'$	Regular type τ is a subtype of τ' in E
$E \vdash \varphi <: \varphi'$	Field type φ is a subtype of φ' in E
$E \vdash a : \tau$	Expression a has type τ in E

The subtyping judgment $E \vdash \tau <: \mathtt{T}$ is used to mean that τ is a well-formed regular type in E, while $E \vdash \varphi <: \mathtt{F}$ means that φ is a well-formed field-type in E. Thus, `T` and `F` also play a role of kinds. For sake of simplicity, we do not allow field variables $\alpha <: \mathtt{F}$ in environments. We have used different meta-variables τ and φ for regular types and field-types for sake of readability, although this is redundant with the constraint enforced by the well-formation rules. The formation of environments is recursively defined with rules for the formation of types and subtyping rules given in figure 2.

The subtyping rules are quite standard. Most of the rules are dedicated to field subtyping; they formally described the relation that was drawn in figure 1. A few facts are worth noticing. First we cannot derive $E \vdash F <: F$. Thus `F` is only used in $E \vdash \varphi <: \mathtt{F}$ to tell that φ is a well-formed field type. It prevents using `F` in object types. The typing rule Sub `PA` is also worth consideration. By transitivity with other rules, it allows $\mathtt{P}\ \tau$ to be a subtype of $\mathtt{M}\ \tau'$, even if types τ and τ' are incompatible. However, it remains true, and this is essential, that $\mathtt{P}\ \tau$ is a subtype of $\mathtt{R}\ \tau'$ if and only if τ is a subtype of τ'.

The rule Sub Obj Invariant describes subtyping for object types. As explained above, row variables are just a meta-notation; thus, the judgment $E \vdash \rho <: \rho'$ is just a short hand for $E \vdash \rho(\ell) <: \rho'(\ell)$ for any label ℓ, which only involves a finite number of them. This rule is restrictive and prevents (positive) occurrences of self to be replaced by $\#\,\tau$ where τ is the current type of the object. In particular, object types cannot be unfolded (see section 6.2).

3.5 Typing rules

Typing rules are given in figure 3. The rules for subsumption, variables, and method invocation are quite standard.

Rule Expr Object has been discussed earlier. The last premise says that the fields ℓ_i may actually be super-types of $\mathtt{P}\ \tau_i$ in ρ and other fields may also be super types of `A`. One cannot simply require that ρ be $(\ell_i : \mathtt{P}\ \tau_i{}^{i \in I}; \mathtt{A})$ and later use subsumption, since the assumption made on the type of x_i while typing a_i could then be too weak.

Rule Expr Update is similar to the overriding rule in [3]. This rule is important since it permits both internal and external updates: the result type of the object is exactly the same as the one before the update.

On the contrary, rule Expr Extend is intended to add new methods that were not necessarily defined before, and thus change the type of the object. There are three different sub-cases in rule Expr Extend; the one that applies is uniquely determined by the given type τ. Then the type of field ℓ in the argument is deduced from the small table.

$$\frac{E \vdash a : \tau \qquad E \vdash \tau <: \tau'}{E \vdash a : \tau'} \text{ (Expr Subsumption)}$$

$$\frac{E, x : \tau, E' \vdash \diamond}{E, x : \tau, E' \vdash x : \tau} \text{ (Expr Var)}$$

(Expr Object) $(\tau \equiv \zeta(\chi)[\rho])$

$$\frac{E, \chi <: \# \tau, x_i : \chi \vdash a_i : \tau_i, i \in I \qquad E, \chi <: \# \tau \vdash (\mathtt{P}\ \tau_i{}^{i \in I} ; \mathtt{A}) <: \rho}{E \vdash \zeta(\chi, \tau)[\ell_i = \varsigma(x_i) a_i{}^{i \in I}] : \tau}$$

(Expr Select)

$$\frac{E \vdash a : \tau \qquad E \vdash \tau <: \zeta(\chi)[\ell : \mathtt{R}^+\ \tau_\ell\ ; \mathtt{U}]}{E \vdash a.\ell : \tau_\ell\{\tau/\chi\}}$$

(Expr Update)

$$\frac{E \vdash a : \tau \qquad E \vdash \tau <: \zeta(\chi)[\ell : \mathtt{R}\ \tau_\ell\ ; \rho_0] \qquad E, \chi <: \# \tau, x : \chi \vdash a_\ell : \tau_\ell}{E \vdash a.\ell \Leftarrow \zeta(\chi, \tau)\varsigma(x) a_\ell : \tau}$$

(Expr Extend) $(\tau \equiv \zeta(\chi)[\rho])$

$$\frac{\begin{array}{c}(\varphi_0, \rho(\ell)) \in \{(\mathtt{A}, \mathtt{P}\ \tau_\ell), (\mathtt{V}\ \tau_\ell, \mathtt{R}\ \tau_\ell), (\mathtt{V}^-\ \tau_\ell, \mathtt{M}^-\ \tau_\ell)\} \\ E \vdash a : \zeta(\chi)[\ell : \varphi_0\ ; \rho \setminus \ell] \qquad E, \chi <: \# \tau, x : \chi \vdash a_\ell : \tau_\ell\end{array}}{E \vdash a.\ell \Leftarrow \zeta(\chi, \tau)\varsigma(x) a_\ell : \tau}$$

Figure 3: Typing rules

Rules Expr Extend and Expr Update both apply only when τ is of the form $\zeta(\chi)[\ell : \mathtt{P}\ \tau_\ell\ ; \rho]$ or $\zeta(\chi)[\ell : \mathtt{R}\ \tau_\ell\ ; \rho]$. Then, the requirements on the type of a are the same (letting the premise of Sub Extend be preceded by a subsumption rule). Thus, different derivations lead to the same judgment. It would also be possible to syntactically distinguish between object extension and method update, as well as to separate the extension between three different primitive corresponding to each of the three typing cases.

3.6 Operational semantics

We give a reduction semantics for a call-by-value strategy. Values are reduced to objects. A leftmost outermost evaluation strategy is enforced by the evaluation contexts C.

$$v ::= \zeta(\chi, \tau)[\ell_i = \varsigma(x_i) a_i{}^{i \in I}] \qquad\qquad C ::= \{\} \mid C.\ell \mid C.\ell \Leftarrow \zeta(\tau, \chi)\varsigma(x) a$$

The reduction rules are given in figure 4. Since programs are explicitly typed, the reduction must also manipulate types in order to maintain programs both well-formed and well-typed, even though it is not type-driven. In fact, the reduction uses an auxiliary binary operation on types $\varphi \Leftarrow \varphi'$, to recompute the witness type of object values during object extension. It is defined in figure 5. The partial $\varphi \Leftarrow \varphi'$ is extended to object types homomorphically, *i.e.*, $\zeta(\chi)[\rho] \Leftarrow \zeta(\chi)[\rho']$ is $\zeta(\chi)[\rho \Leftarrow \rho']$. Type extension is defined so as it validates lemma 4. When there is some flexibility, we sought for more uniformnity. Type extension is undefined when the cell is left empty in the figure. Those are cases that will never meet the hypotheses of lemma 4.

Let ℓ and $\ell_i{}^{i \in I}$ be distinct labels, j in I,
and v be of the form $\zeta(\chi, \tau)[\ell_i = \varsigma(x_i) a_i{}^{i \in I}]$.

$$v.\ell_j \longrightarrow a_j\{\tau/\chi\}\{v/x\} \qquad \text{(Select)}$$

$$v.\ell_j \Leftarrow \zeta(\chi, \tau')\varsigma(x)a \longrightarrow \zeta(\chi, \tau \Leftarrow \tau')[\ell_i = \varsigma(x_i)a_i{}^{i \in I - j}, \ell_j = \varsigma(x)a] \qquad \text{(Update)}$$

$$v.\ell \Leftarrow \zeta(\chi, \tau')\varsigma(x_0)a_0 \longrightarrow \zeta(\chi, \tau \Leftarrow \tau')[\ell_i = \varsigma(x_i)a_i{}^{i \in I}, \ell = \varsigma(x)a] \qquad \text{(Extend)}$$

$$\text{if } a_1 \longrightarrow a_2 \text{ then } C\{a_1\} \longrightarrow C\{a_2\} \qquad \text{(Context)}$$

Figure 4: Reduction rules

<table>
<tr><td colspan="2" rowspan="2">$\Leftarrow$</td><td colspan="7">φ'</td><td>$\#\varphi$</td></tr>
<tr><td>$\mathtt{P}\ \tau'$, $\mathtt{A}$</td><td>$\mathtt{U}$</td><td>$\mathtt{R}\ \tau'$, $\mathtt{R}^+\ \tau'$</td><td>$\mathtt{M}\ \tau'$</td><td>$\mathtt{M}^-\ \tau'$</td><td>$\mathtt{V}\ \tau'$</td><td>$\mathtt{V}^-\ \tau'$</td><td></td></tr>
<tr><td rowspan="5">φ</td><td>$\mathtt{P}\ \tau$, $\mathtt{A}$</td><td colspan="7">φ'</td><td>$\mathtt{U}$</td></tr>
<tr><td>$\mathtt{R}\ \tau$, $\mathtt{R}^+\ \tau$, $\mathtt{U}$, $\mathtt{M}^-\ \tau$</td><td>.</td><td colspan="6">φ</td><td>φ</td></tr>
<tr><td>$\mathtt{M}\ \tau$</td><td>.</td><td>.</td><td>$\mathtt{R}\ \tau$</td><td colspan="2">φ</td><td>$\mathtt{V}\ \tau$</td><td>φ</td><td>φ</td></tr>
<tr><td>$\mathtt{V}\ \tau$</td><td>.</td><td>.</td><td colspan="3">$\mathtt{R}\ \tau$</td><td colspan="2">φ</td><td>$\mathtt{R}\ \tau$</td></tr>
<tr><td>$\mathtt{V}^-\ \tau$</td><td>.</td><td>.</td><td>.</td><td>.</td><td>$\mathtt{M}^-\ \tau$</td><td>.</td><td>φ</td><td>$\mathtt{M}^-\ \tau$</td></tr>
<tr><td colspan="2">$\#\varphi'$</td><td>$\mathtt{U}$</td><td colspan="4">φ'</td><td>$\mathtt{R}\ \tau'$</td><td>$\mathtt{M}^-\ \tau'$</td><td></td></tr>
</table>

Figure 5: Type reduction $\varphi \Leftarrow \varphi'$

4 Soundness of the typing rules

The soundness of the typing rules results from a combination of subject reduction and canonical forms. The proof of subject reduction is standard (see [3] for instance). A few classical lemmas help simplifying the main proof.

Lemma 1 (Bound weakening) *If $E \vdash \tau <: \tau'$ and $E, \alpha <: \tau', E' \vdash \mathcal{J}$, then $E, \alpha <: \tau, E' \vdash \mathcal{J}$.*

Proof: By induction on the size of the proof of the derivation of the second. ■

Lemma 2 (Substitution)

1. *If $E, \alpha <: \tau, E' \vdash \mathcal{J}$ and $E \vdash \tau' <: \tau$, then $E, E'\{\tau'/\alpha\} \vdash \mathcal{J}\{\tau'/\alpha\}$.*
2. *If $E, x : \tau, E' \vdash \mathcal{J}$ and $E \vdash a : \tau$, then $E, E' \vdash \mathcal{J}\{a/x\}$.*

Lemma 3 (Structural subtyping)

1. *If $\tau \equiv \zeta(\chi)[\rho]$ and $E \vdash \tau <: \tau'$, then τ' is either $\mathtt{T}$ or of the form $\zeta(\chi)[\rho']$ and $E, \chi <: \mathtt{T} \vdash \rho <: \rho'$.*
2. *If $E \vdash \varphi <: \mathtt{R}\ \tau_\ell$, then φ is either $\mathtt{R}\ \tau_l$ or $\mathtt{P}\ \tau_0$ where $E \vdash \tau_0 <: \tau_\ell$.*

3. *If* $E \vdash \varphi <: \mathtt{R}^{+}\,\tau_\ell$*, then* φ *is either* $\mathtt{P}\;\tau_0$*,* $\mathtt{R}\;\tau_0$*, or* $\mathtt{R}^{+}\,\tau_0$ *where* $E \vdash \tau_0 <: \varphi_\ell$*.*

4. *If* $E \vdash \varphi <: \mathtt{P}\;\tau_\ell$*, then* φ *is* $\mathtt{P}\;\tau_0$ *where* $E \vdash \tau_0 <: \tau_\ell$*.*

   Etc.

Proof: By induction on the size of subtyping derivations. Should use the fact that transitivity rules can be pushed to the leaves. ∎

The proof of subject reduction also uses an essential lemma that relates computation on types to subtyping. Actually, the proof does not depend on the particular definition of #, but only on the following lemma.

Lemma 4 (Type computation) *Let* τ *and* τ' *be two object types* $\zeta(\chi)[\rho]$ *and* $\zeta(\chi)[\rho']$*. Assume that there exists a row* ρ'' *such that* $E, \chi <: \mathtt{T} \vdash \rho <: \rho''$ *and for each label* ℓ*, the pair* $(\rho''(\ell), \rho'(\ell))$ *is one of the four forms* $(\mathtt{A}, \mathtt{P}\;\tau_\ell)$*,* $(\mathtt{V}\;\tau_\ell, \mathtt{R}\;\tau_\ell)$*,* $(\mathtt{V}^-\,\tau_\ell, \mathtt{M}^-\,\tau_\ell)$*, or* (φ, φ)*. Let* $\hat\tau$ *be* $\tau \Leftarrow \tau'$ *and* $\hat\rho$ *be* $\rho \Leftarrow \rho'$*. Then,*

$$E \vdash \hat\tau <: \tau' \qquad\qquad E \vdash \#\,\hat\tau <: \#\,\tau \qquad\qquad E \vdash \#\,\hat\tau <: \#\,\tau'$$

Moreover, in the three first cases, if $E, \chi <: \mathtt{T} \vdash \rho(\ell) <: \rho'(\ell)$*, then* $E, \chi <: \mathtt{T} \vdash \rho(\ell) <: \hat\rho(\ell)$*; otherwise* $E, \chi <: \mathtt{T} \vdash \mathtt{P}\;\tau_\ell <: \hat\rho(\ell)$*.*

Lemma 5 (Virtual methods) *If* $E \vdash \tau <: \zeta(\chi)[\mathtt{U}]$*, then* $E \vdash \tau <: \#\,\tau$*.*

Proof: This is obviously true field by field: the only field that does not satisfy $E \vdash \tau <: \#\,\tau$ are virtual fields, which are excluded if $E \vdash \tau <: \mathtt{U}$. The property easily follows for object types. ∎

Theorem 1 (Subject Reduction) *Typings are preserved by reduction. If* $E \vdash a : \tau_a$ *and* $a \longrightarrow a'$ *then* $E \vdash a' : \tau_a$*.*

Theorem 2 (Canonical Forms) *Well-typed expressions that cannot be reduced are values. If* $\emptyset \vdash a : \tau$ *and there exists no* a' *such that* $a \longrightarrow a'$*, then* a *is a value.*

Proof: If a value v has type $\zeta(\chi)[\ell{:}\,\mathtt{P}\;\tau\;;\,\mathtt{U}]$, then v must have a field ℓ. The theorem is then a trivial induction on the size of a, assuming that a cannot be reduced. ∎

5 Examples

For simplicity, we assume that the core calculus has been extended with abstraction and application. This extension could either be primitive or derived from the encoding given in section 5.6. For brievity, we write $a.\ell \Leftarrow a'$ instead of $a.\ell \Leftarrow \zeta(\chi, \tau)\varsigma(z)a'$ when a' does not depend on the self parameter z. In practice, other abbreviations could be made, but we avoid them here to reduce confusion.

We consider the simple example of points and colored points. These objects can of course already be written in [3]. The expressiveness of our calculus is not so much its capability to write new forms of complete objects but to provide new means of defining them. This provides more flexibility, increases security in several ways , and removes the complexity of the encoding of classes into objects.

5.1 Objects

A point object p_0 can be defined as follows:

$$\zeta(\chi, \texttt{point})[x = 0 \; ; mv = \varsigma(z)\lambda y.(z.x \Leftarrow y) \; ; print = \varsigma(z) print_int\ z.x]$$

where point is $\zeta(\chi)[x\!:\!\texttt{R int} \; ; mv\!:\!\texttt{P int} \to \chi \; ; print\!:\!\texttt{P unit} \; ; \texttt{A}]$. As in [3], new points can be created using method update as in $p_0.x \Leftarrow \zeta(\chi, \texttt{point})\varsigma(z)1$. Moreover, colored points can be defined inheriting from points:

$$\texttt{cpoint} \triangleq \zeta(\chi)[x\!:\!\texttt{R int} \; ; c\!:\!\texttt{R bool} \; ; mv\!:\!\texttt{P int} \to \chi \; ; print\!:\!\texttt{P unit} \; ; \texttt{A}]$$

$$cp \triangleq (p_0.c \Leftarrow \zeta(\chi, \texttt{cpoint})\varsigma(z)\texttt{true}).print \Leftarrow \zeta(\chi, \texttt{cpoint})\varsigma(z)\texttt{if}\ z.c\ \texttt{then}\ print_int\ z.x$$

When two values of different types have a common super-type τ, they can be interchanged at type τ. Here, cpoint is not a subtype of point, since both types carry too precise type information. However, they admit the common super-type $\zeta(\chi)[x\!:\!\texttt{R int} \; ; c\!:\!\texttt{U} \; ; mv\!:\!\texttt{P int} \to \chi \; ; print\!:\!\texttt{P unit} \; ; \texttt{A}]$..

5.2 Abstraction via subtyping

Subtyping can also be used to enforce security. For instance, field x may be hidden by weakening its type to U. Similarly, method mv may be protected against further redefinition by weakening its type to $\texttt{R}^+\,\tau$. That is, by giving p_0 the type $\zeta(\chi)[mv\!:\!\texttt{R}^+\texttt{int} \to \chi \; ; print\!:\!\texttt{R unit} \; ; \texttt{U}]$. While method mv can no longer be directly redefined, there is still a possibility for indirect redefinition. For instance, method $print$ could have been written so that it overrides method mv before printing. To ensure that a method can never be redefined, directly or indirectly, it must be given type $\texttt{R}^+\,\tau$ at its creation.

5.3 Virtual methods

The creation of new points by updating the field of an already existing point is not quite satisfactory since it requires the use of default methods to represent the undefined state, which are often arbitrary and may be a source of errors. Indeed, a class of points can be seen as a virtual point lacking its field components.

$$\texttt{POINT} \triangleq \zeta(\chi)[x\!:\!\texttt{V int} \; ; mv\!:\!\texttt{P int} \to \chi \; ; print\!:\!\texttt{P unit} \; ; \texttt{A}]$$

$$P \triangleq \zeta(\chi, \texttt{POINT})[mv = \varsigma(z)\lambda y.(z.x \Leftarrow y) \; ; print = \varsigma(z) print_int\ z.x]$$

New points are then created by filling in the missing fields:

$$\texttt{new_point} \triangleq \lambda y.(P.x \Leftarrow \zeta(\chi, \texttt{point})\varsigma(z)y) \qquad p_1 \triangleq \texttt{new_point}\ 0$$

5.4 Traditional class-based perspective

To keep closer to the traditional approach, we may by default choose to hide both fields corresponding to instance variables and the extendible capabilities of the remaining methods. For instance, treating x as an instance variable, and mv and $print$ as "regular" methods, we choose $\zeta(\chi)[mv\!:\!\texttt{R}^+\texttt{int} \to \chi \; ; print\!:\!\texttt{R}^+\texttt{unit} \; ; \texttt{U}]$ for point. Intuitively, the object-type point hides all information that is not necessary to increase security. Conversely, the class-type POINT remains as

precise as possible, to keep expressiveness. Indeed, a class of points is still an object. However, as opposed to the previous section, we adopt some uniform, more structured style, treating "real" objects differently from those representing classes.

In colored points, we may choose to leave field c readable and overridable, as if we defined two methods *set_c* and *get_c*.

$$\texttt{cpoint} \stackrel{\triangle}{=} \zeta(\chi)[c{:}\,\texttt{R}\,\texttt{bool}\,;\, mv{:}\,\texttt{R}^{+}\texttt{int} \to \chi\,;\, print{:}\,\texttt{R}^{+}\texttt{unit}\,;\,\texttt{U}]$$

Single inheritance is obtained by class extension:

$$\texttt{CPOINT} \stackrel{\triangle}{=} \zeta(\chi)[x{:}\,\texttt{V}\,\texttt{int}\,;\, c{:}\,\texttt{V}\,\texttt{bool}\,;\, mv{:}\,\texttt{P}\,\texttt{int} \to \chi\,;\, print{:}\,\texttt{P}\,\texttt{unit}\,;\,\texttt{A}]$$

$$CP \stackrel{\triangle}{=} (P.print \Leftarrow \zeta(\chi, \texttt{CPOINT})\varsigma(z)\texttt{if}\ z.c\ \texttt{then}\ print_int\ z.x)$$

$$\texttt{new_cpoint} \stackrel{\triangle}{=} \lambda y.\lambda w.(CP.x \Leftarrow \zeta(\chi, \texttt{cpoint})\varsigma(z)y).c \Leftarrow \zeta(\chi, \texttt{cpoint})\varsigma(z)w$$

While CPOINT is not a subtype of POINT at the class level, we recover the usual relationship that cpoint is a subtype of point at the object level. Moreover, at the object level, types are invariant by #. Thus, we also recover the subtyping relation of [3]. In particular, object types can be unfolded.

5.5 An advanced example

A colorable point p' is a point prepared to be colored without actually being colored. It can be obtained by adding to the point p_0 an extra method paint that when called with an argument y returns the colored point obtained by adding the color field c with value y and by updating the print method of p_0.

$$p' \stackrel{\triangle}{=} p_0.paint \Leftarrow \varsigma(z, \texttt{point}')\lambda y.$$
$$((z.c \Leftarrow y).print \Leftarrow \zeta(\chi, \texttt{cpoint})\varsigma(z)\texttt{if}\ z.c\ \texttt{then}\ print_int\ z.x)$$

where point′ is

$$\zeta(\chi)[x{:}\,\texttt{R}\,\texttt{int}\,;\, ...print{:}\,\texttt{R}\,\texttt{unit}\,;\, paint{:}\,\texttt{P}\,\texttt{bool} \to \texttt{cpoint}\,;\, c{:}\,\texttt{M}\,\texttt{bool}\,;\,\texttt{U}]$$

This example may be seen as the installation (method *paint*) of a new behavior (method *print*) that interacts with the existing state x and adds some new state c. The above solution becomes more interesting if each installation involves many methods, and especially if several installation are either different fields of the same objects or the same field of different objects. Then, the installation procedure can be selected dynamically by message invocation instead of manually by applying an external function to the object.

5.6 Encoding of the lambda-calculus

This part improves the encoding proposed in [3]. It also illustrates the use of virtual methods and variances. The untyped encoding of the lambda-calculus into objects in [3] is the following[2]:

$$\langle\!\langle x \rangle\!\rangle \stackrel{\triangle}{=} x.\texttt{arg} \qquad \langle\!\langle \lambda x.M \rangle\!\rangle \stackrel{\triangle}{=} [\texttt{arg} = \varsigma(x)x.\texttt{arg}\,;\,\texttt{val} = \varsigma(x).\langle\!\langle M \rangle\!\rangle]$$

$$\langle\!\langle M\ M' \rangle\!\rangle \stackrel{\triangle}{=} (\langle\!\langle M \rangle\!\rangle.\texttt{arg} \Leftarrow \varsigma(x)\langle\!\langle M' \rangle\!\rangle).\texttt{val}$$

[2]If both functions and objects co-exist, one should actually mark variables introduced by the encoding of functions so as to leave the other variables unchanged.

A function is encoded as an object with a diverging method arg. The encoding of an application overrides the method arg of the encoding of the function with the encoding of the argument and invokes the method val of the resulting object. Programs obtained by the translation of functional programs will never call val before loading the argument. However, if the encoding is used as a programming style, the type system will not provide as much safety as a type system with primitive function types would. The method val could also be called, accidently, before the field arg has been overridden. In general, this will, in turn, call the method arg and diverge. The use of default diverging methods is a hack that palliates the absence of virtual methods. It can be assimilated to a "method not understood" type error and one could argue that the encoding of [3] is not strongly typed.

The encoding can be improved using object extension to treat a function $\lambda x.M$ as an object $[\mathtt{val} = \varsigma(x).\langle\!\langle M \rangle\!\rangle]$ with a virtual method arg (remember that $x.\mathtt{arg}$ may appear in $\langle\!\langle M \rangle\!\rangle$). The type-system will then prevent the method val to be called before the argument has been loaded. More precisely, let us consider the simply typed lambda-calculus:

$$t ::= \alpha \mid t \to t \qquad\qquad M ::= x \mid \lambda x : t.M \mid M\ M$$

Functional types are encoded as follows:

$$\langle\!\langle \alpha \rangle\!\rangle \triangleq \alpha \qquad\qquad \langle\!\langle t \to t' \rangle\!\rangle \triangleq \zeta(\chi)[\mathtt{arg}\colon \mathtt{V}^{-}\langle\!\langle t \rangle\!\rangle\ ;\ \mathtt{val}\colon \mathtt{R}^{+}\langle\!\langle t' \rangle\!\rangle\ ;\ \mathtt{U}]$$

This naturally induces a subtyping relation between function types that is contra-variant on the domain and covariant on the co-domain. The typed encoding is given by the following inference rules:

$$\frac{x : t \in A}{A \vdash x : t \Rightarrow x.\mathtt{arg}} \qquad\qquad \frac{A, x : t \vdash M : t' \Rightarrow a \qquad x \notin dom(A)}{A \vdash \lambda x : t.M : t \to t' \Rightarrow \zeta(\chi, \langle\!\langle t \to t' \rangle\!\rangle)[\mathtt{val} = \varsigma(x).a]}$$

$$\frac{A \vdash M : t' \to t \Rightarrow a \qquad A \vdash M' : t' \Rightarrow a'}{A \vdash M\ M' : t \Rightarrow (a.\mathtt{arg} \Leftarrow \zeta(\chi, \#\langle\!\langle t \to t' \rangle\!\rangle)\varsigma(x)a').\mathtt{val}}$$

It is easy to see that the translation transforms well-typed judgments $\emptyset \vdash M : t$ into well-typed judgments $\emptyset \vdash \langle\!\langle M \rangle\!\rangle : \langle\!\langle t \rangle\!\rangle$.

As in [3], the translation provides a call-by-name operational semantics for the lambda-calculus. The encoding of [3] also provides an equational theory for the object calculus and, thefore, for the lambda calculus, via translation, which we do not.

6 Discussion

6.1 Variations

Several variations can be made by consistently modifying field-types, their subtyping relationship, and the typing rule for object extension. The easiest is to drop some subtyping asumption (such as SUB PP, or SUB PA) or drop the field-type $\mathtt{P}\ \tau$ altogether. This weakens the type system (some examples are not typable any longer), but it retains the essential features. More significant simplifications can be made at the price of a higher restriction of expressiveness. For instance, virtual field-types could be removed.

Some extensions or modifications to the type hierarchy are also possible. For instance, one could introduce fields of type $\dagger\mathtt{P}\ \tau$ that do no depend on any other method. These methods would

be dual of those of type $\mathtt{P}\ \tau$ on which no other method depend; somehow they would behave as record fields in the sense they could always be called even if the object is virtual. This extends to field-types $\dagger\mathtt{R}\ \tau$ and $\dagger\,\mathtt{R}^{+}\ \tau$ similarly.

6.2 Better subtyping for object types

The subtyping rule Sub-Obj-Invariant does not allow unfolding of object types. It is thus weaker that the Abadi-Cardelli:

$$\text{(Sub Obj Deep) } (\tau \equiv \zeta(\chi)[\rho], \tau' \equiv \zeta(\chi)[\rho'])$$
$$\frac{E \vdash \tau <: \mathtt{T} \qquad E \vdash \tau' <: \mathtt{T} \qquad E, \chi <: \tau \vdash \rho <: \rho'}{E \vdash \tau <: \tau'}$$

This rule would not be correct, since it would not be transitive. Indeed transitivity would require that $A \vdash \tau <: \tau'$ implies $A \vdash \#\,\tau <: \#\,\tau'$ which is not true.

Just replacing the bound $\mathtt{T}$ of χ in Sub Obj Deep by $\#\,\tau$ would actually not behave well with respect to transitivity. In a preliminary version of this work [26], we added another premise to recover transitivity. However, this simultaneously weakens the subtyping relationship, and some useful examples become untypable.

It should be possible to define a subtyping rule that allows unfolding of self types only when objects have no more extension capabilities. It seems however, that the subtyping structure of fields should either be simplified (e.g. eliminating the arrow from $\mathtt{M}\ \tau$ to $\mathtt{V}\ \tau$) or almost equivalently enriched to avoid $\mathtt{M}\ \tau <: \mathtt{V}\ \tau$ but only once in a certain definite state.

6.3 Extensions

Imperative update is an orthogonal issue to the one studied here, and it could be added without any problem. Object extension should, of course, remain functional.

Equational theory We see no difficulty in adding an equational theory to our calculus, but this remains to be investigated. Treating object extension as a commutative operator would allow to reduce object construction to a sequence of object extensions of the empty object (virtual methods would be crucial here).

Higher-order types As shown above, our objects are sufficiently powerful to represent classes. As opposed to [3], this does not necessitate higher-order polymorphism because methods are already required to be parametric in all possible extensions of self.

The addition of higher-order polymorphism might still be useful, in particular to enable parametric classes. We believe that there is no problem in constraining type abstraction by some supertype bound, written $\alpha <: \tau$ as in $F_{<:}$. However, it would also be useful to introduce #-bounds of the form $\alpha <: \#\,\tau$. This might require more investigation.

Row variables and binary methods We have used row variables only as a meta-notation for simplifying the presentation. It would be interesting to really allow row variables in types. This would probably augment the expressiveness of the language, since it should provide some form of matching that revealed quite useful, especially for binary methods [11, 9, 27].

Actually, it remains to investigate how the presented calculus could be extended to cope with binary methods. Row variables might not be sufficient to express matching, and some new form of

matching might have to be found. It is unclear whether the known solutions [10] could be adapted to our calculus.

7 Comparison with other works

Our proposal is built on the calculus of objects of Abadi and Cardelli [3], which is invoked throughout this paper. Our use of variance annotations is in principle similar to theirs. By attaching variance annotations to field-types rather than to fields themselves, we eliminate some useless types such as $\mathtt{M}^{+}\,\tau$. Indeed, such a field could not be overriden, nor accessed, and thus it could be just given type $\mathtt{U}$. (Our use of variances also eliminate the ability to specify the type of a field without specifying its variance, which may cause problem with type inference [22].) An essential imported tool is the structure of record-types of [23], which was originally designed for type inference in ML [24]. The use of a richer structure of record types has previously been proposed for type checking records [15, 16, 13, 12]. To our knowledge, the benefits of symmetric information were first transfered from record types to object types in [25]. There, first-order typing rules for objects with extension and both deep and width subtyping were roughly drafted without any formal treatment.

A similar approach has also been independently proposed by Bono, Liquori and others. Their first related work [6] has later lead to many closely related proposals [8, 7, 4, 18, 17, 19]. Most of these are extensions of the Fisher-Honsel calculus of objects [20]. The differences between their approach and the one of [3] (which is also ours) are not always significant but they make a close comparisson more difficult. Only two of these works [19, 18] are extensions of the Abadi-Cardelli calculus of objects [3] and are thus more connected to our proposal. The first-order version [19], is subsumed by both [25] (which also covers deep subtyping) and [18] (which also addresses self types.) Our proposal extends both [25] and [18].

The most interesting comparison can be made with [18]. The main motivation and the key idea behind both proposals are similar: they integrate object subtyping and object extension, using a richer type structure to preserve type soundness. Saturated *vs* diamond types correspond to our object-types with a field template $\mathtt{U}$ *vs* $\mathtt{A}$, respectively. Our treatment seems more uniform. We only have one kind of object types. We distinguish between the "saturation" and "diamond" properties in fields instead of objects. As a result, we can write an object type that is saturated, except for a few particular fields. Our proposal also includes several additional features: it addresses deep subtyping and virtual methods; it also allows methods to extend self. Moreover, in our proposal, the subtyping relationship is structural for object types. Additionally, subtyping axioms are only given at the level of fields, each one of them treating a different important subtyping capability. As a result, object types have a more regular structure, and can easily be adapted to further extensions. We think this is easier to understand, to modify, and to manipulate.

An alternative to virtual methods has also been studied in [4], using a quite different approach, which consists in annotating each method with the list of all other methods they depend on. Thus, each method has a different view of self. Their approach to incomplete objects is, in principle, more powerful that ours; in particular, they can type programs that even traditional class-based languages would reject. We found their types of objects too detailed, and thus their proposal less practical than ours. (Tracing dependencies is closer to some form of program analysis than to standard type systems.) In fact, we intendedly restricted our type system so that methods have a uniform view of self. In practice, our solution is sufficient to capture common forms of inheritance.

In [20], pre-objects have pro-types and can be turned into objects with obj-types by subtyping. Pro-types and obj-types are similar to our object types $\zeta(\chi)[\ell_i\colon \mathtt{R}\ \tau_i\,^{i\in I}\ ;\mathtt{A}]$ and $\zeta(\chi)[\ell_i\colon \mathtt{R}^{+}\ \tau_i\,^{i\in I}\ ;\mathtt{U}]$. One difference is that, in our case, subtyping is defined and permitted field by field rather than all at

once. Fisher and Mitchell also studied the relationship between objects and classes in [14]. They use bounded existential quantification to hide some of the structure of the object in the *public* interface. This still allows public methods to be called, while *private* methods become innaccessible. In our calculus, the richer structure of objects permits to use subtyping instead of bounded existential quantification to provide a similar abstraction. This is not suprising, theoretically, since subtyping, as existential quantification, is a lost of type information. However, this is practically a significant difference, since subtyping allows more explicit type information but is less expressive. Another difference is that using the standard record types they had to introduce record sorts to express negative type information. As pointed out in a more recent paper [5], the design of the language of kinds becomes important for modularity. In particular, [5] improves over [14] by changing default kinds from unknown (`U` in our setting) to absent (`A`). Instead, our record types express positive and negative information symmetrically and are viewed as total functions from fields to types, which avoids the somehow *ad hoc* language of sorts.

In a recent paper, Riecke and Stone have circumvented the problem of merging extension with deep and width subtyping by changing the semantics of objects [28]. In fact, their semantics remain in correspondance with the standard semantics of objects in the general case, but the semantics of extension is changed so that the counter example becomes sound in the new semantics. They distinguish between method update and object extension. Then, a field that is already defined is automatically renamed by extension into an anonymous field that becomes externally inaccessible.

With their semantics, some of our enriched type information would become obsolete for ensuring type soundness, but it might remain useful for compile-time optimizations. Other pieces of information, e.g. virtual types, would remain quite pertinent.

8 Conclusion

We have proposed a uniform and flexible method for enriching type systems of object calculi by refining the field structure of object types, so that they carry more precise type information.

Applying our approach to the object calculus of Abadi and Cardelli, we have integrated object extension and depth and width subtyping, with covariant final methods and contra-variant virtual methods, in a type-safe calculus. When sufficient type information is revealed, objects may represent classes. Type information may also be hidden progressively, until objects can be used and interchanged in a traditional fashion.

An important gain is to avoid the encoding of classes as records of pre-methods. Instead, we provide a more uniform, direct approach. Another benefit of this integration is to allow mixed formed of classes and objects. The use of richer object types also increases both safety by capturing more dynamic misbehavior as static type errors and security by allowing more privacy via subtyping. Moreover, our approach subsumes several other unrelated proposals, and it might provide a unified framework for studying or comparing new proposals. Some extensions and variations are clearly possible, provided the operations on objects, their types and the subtyping hierarchy are changed consistently.

More investigation still remains to be done. Adding an equational theory to the calculus, would simplify our primitives, since objects could always be built field by field using object extension only. This might also be a first step towards a better integration of record-based and delegation-based object calculi. In the future, we would also like to study the potential increase of expressiveness that field and row variables could provide. Of course, investigating binary methods remains one of the most important issues.

Classes can be viewed as objects. We hope that an even richer type structure would finally

enable to see objects for what they really are —records of functions— in the (yet untyped) self-application interpretation.

References

[1] Martín Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. In *Theoretical Aspects of Computer Software*, pages 296–320. Springer-Verlag, April 1994.

[2] Martín Abadi and Luca Cardelli. A theory of primitive objects: Second-order systems. *Science of Computer Programming*, 25(2-3):81–116, December 1995. Preliminary version appeared in D. Sanella, editor, Proceedings of European Symposium on Programming, pages 1-24. Springer-Verlag, April 1994.

[3] Martín Abadi and Luca Cardelli. *A theory of objects*. Springer, 1996.

[4] V. Bono, M. Bugliesi, M. Dezani-Ciancaglini, and L. Liquori. Subtyping Constraints for Incomplete Objects. In *Proceedings of TAPSOFT-CAAP-97, International Joint Conference on the Theory and Practice of Software Development*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[5] V. Bono and K. Fisher. An imperative, first-order calculus with object extension. In *Informal Proceedings of the FOOL 5 workshop on Foundations of Object Oriented Programming*, Sans Diego, CA, January 1998. To appear.

[6] V. Bono and L. Liquori. A subtyping for the fisher-honsell-mitchell lambda calculus of object. In *Proc. of CSL-94, International Conference of Computer Science Logic*, volume 933 of *Lecture Notes in Computer Science*, pages 16–30. Springer-Verlag, 1995.

[7] Viviana Bono and Michele Bugliesi. A lambda calculus of incomplete objects. In *Proceedings of Mathematical Foundations of Computer Science(MFCS)*, number 1113 in Lecture Notes in Computer Science, pages 218–229, 1996.

[8] Viviana Bono and Michele Bugliesi. Matching constraints for the lambda calculus of objects. In *Proceedings of (MFCS)*, 1997.

[9] Kim B. Bruce. Typing in object-oriented languages: Achieving expressibility and safety. Revised version to appear in Computing Surveys, November 1995.

[10] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, Valery Trifonov) the Hopkins Objects Group (Jonathan Eifrig, Scott Smith, Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1996.

[11] Kim B. Bruce, Angela Schuett, and Robert van Gent. Polytoil: A type-safe polymorphic object-oriented language. In *ECOOP*, number 952 in LNCS, pages 27–51. Springer Verlag, 1995.

[12] Luca Cardelli. Extensible records in a pure calculus of subtyping. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*, pages 373–425. MIT Press, 1994.

[13] Luca Cardelli and John C. Mitchell. Operations on records. In *Fifth International Conference on Mathematical Foundations of Programming Semantics*, 1989.

[14] K. Fisher and J. C. Mitchell. On the relationship between classes, objects and data abstraction. *Theoretical And Practice of Objects Systems*, To appear, 1998. A preliminary version appeared in the proceedings of the International Summer School on Mathematics of Program Construction, Marktoberdorf, Germany, Springer LNCS, 1997.

[15] Robert W. Harper and Benjamin C. Pierce. Extensible records without subsumption. Technical Report CMU-CS-90-102, Carnegie Mellon University, Pittsburg, Pensylvania, February 1990.

[16] Robert W. Harper and Benjamin C. Pierce. A record calculus based on symmetric concatenation. Technical Report CMU-CS-90-157, Carnegie Mellon University, Pittsburg, Pensylvania, February 1990.

[17] L. Liquori and G. Castagna. A Typed Lambda Calculus of Objects. In *Proc. of ASIAN-96, International Asian Computing Science Conference*, volume 1212 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[18] Luigi Liquori. Bounded polymorphism for extensible objects. Technical Report CS-24-96, Dipartimento di Informatica, Universita' di Torino, 1997.

[19] Luigi Liquori. An Extended Theory of Primitive Objects: First Order System. In *Proceedings of ECOOP-97, International European Conference on Object Oriented Programming*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[20] John C. Mitchell and Kathleen Fisher. A delegation-based object calculus with subtyping. In *Fundamentals of Computation Theory*, number 965 in LNCS, pages 42–61. Springer, 1995.

[21] John C. Mitchell, Furio Honsell, and Kathleen Fisher. A lambda calculus of objects and method specialization. In *IEEE Symposium on Logic in Computer Science*, pages 26–38, June 1993.

[22] Jens Palsberg and Trevor Jim. Type inference of object types with variances. Private Discussion, 1996.

[23] Didier Rémy. Syntactic theories and the algebra of record terms. Research Report 1869, Institut National de Recherche en Informatique et Automatisme, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, 1993.

[24] Didier Rémy. Type inference for records in a natural extension of ML. In Carl A. Gunter and John C. Mitchell, editors, *Theoretical Aspects Of Object-Oriented Programming. Types, Semantics and Language Design*. MIT Press, 1993.

[25] Didier Rémy. Better subtypes and row variables for record types. Presented at the workshop on Advances in types for computer science at the Newton Institute, Cambridge, UK, August 1995.

[26] Didier Rémy. From classes to objects via subtyping. In *European Symposium On Programming*, volume 1381 of *Lecture Notes in Computer Science*. Springer, March 1998.

[27] Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theoretical And Practice of Objects Systems*, To appear, 1998. A preliminary version appeared in the proceedings of the 24th ACM Conference on Principles of Programming Languages, 1997.

[28] Jon G. Riecke and Christopher A. Stone. Privacy via subsumption. In *Informal Proceedings of the FOOL 5 workshop on Foundations of Object Oriented Programming*, Sans Diego, CA, January 1998. To appear.

A Type computation

Lemma 4 (Type computation) *Let τ and τ' be two object types $\zeta(\chi)[\rho]$ and $\zeta(\chi)[\rho']$. Assume that there exists a row ρ'' such that $E, \chi <: \mathtt{T} \vdash \rho <: \rho''$ and for each label ℓ, the pair $(\rho''(\ell), \rho'(\ell))$ is one of the four forms $(\mathtt{A}, \mathtt{P}\ \tau_\ell)$, $(\mathtt{V}\ \tau_\ell, \mathtt{R}\ \tau_\ell)$, $(\mathtt{V}^-\ \tau_\ell, \mathtt{M}^-\ \tau_\ell)$, or (φ, φ). Let $\hat\tau$ be $\tau \Leftarrow \tau'$ and $\hat\rho$ be $\rho \Leftarrow \rho'$. Then,*

$$E \vdash \hat\tau <: \tau' \qquad E \vdash \#\,\hat\tau <: \#\,\tau \qquad E \vdash \#\,\hat\tau <: \#\,\tau'$$

Moreover, in the three first cases, if $E, \chi <: \mathtt{T} \vdash \rho(\ell) <: \rho'(\ell)$, then $E, \chi <: \mathtt{T} \vdash \rho(\ell) <: \hat\rho(\ell)$; otherwise $E, \chi <: \mathtt{T} \vdash \mathtt{P}\ \tau_\ell <: \hat\rho(\ell)$.

Proof: Let E' be $E, \chi <: \mathtt{T}$. By rule Sub Obj Invariant it sufficies to check

$$(1)\ E' \vdash \hat\rho <: \rho' \qquad (2)\ E' \vdash \#\,\hat\rho <: \#\,\rho \qquad (3)\ E' \vdash \#\,\hat\rho <: \#\,\rho'$$

$$(4)\ \begin{cases} E' \vdash \rho <: \hat\rho & \text{if } \rho'' \text{ is } \rho', \\ \exists \tau'_\ell, E' \vdash \tau_\ell <: \tau'_\ell \wedge E' \vdash \mathtt{P}\ \tau'_\ell <: \hat\tau(\ell) & \text{otherwise.} \end{cases}$$

independently for each cell of the table 5 and for any of the fourth possible forms (three first forms for (4)).

Case (φ): These cases cannot occur because all hypotheses cannot be met simultaneously.

Case first line: Note that this completely covers the case where (φ'', φ') is $(\mathtt{A}, \mathtt{P}\ \tau_\ell)$. Properties (1) and (3) are immediate since $\hat\varphi$ is φ' and (2) is obvious since $\#\,\varphi$ is $\mathtt{U}$. Since $\hat\varphi$ is φ', $E' \vdash \varphi <: \hat\varphi$ follows from $E' \vdash \varphi <: \varphi'$, and $E' \vdash \varphi' <: \hat\varphi$ is always true, hence (4).

Case $E' \vdash \varphi <: \varphi'$: In particular, this covers the case where (φ'', φ') is (φ, φ).

Subcase $\hat\varphi$ is φ: Then (1), (2) and (4) are obvious. When $\#\,\varphi'$ is φ', it happens that $\#\,\varphi$ is also φ, thus (3) is true. There are 6 remaining cases in the last two columns:

- In the last column, we must show that $E' \vdash \#\,\varphi <: \mathtt{M}^-\ \tau'$. If φ is $\mathtt{V}^-\ \tau$ or $\mathtt{M}^-\ \tau$, then $E \vdash \tau' <: \tau$, and since $\#\,\varphi$ is $\mathtt{M}^-\ \tau$, then $E' \vdash \#\,\varphi <: \mathtt{V}^-\ \tau'$. Otherwise, φ is either $\mathtt{R}\ \tau$ or $\mathtt{M}\ \tau$, invariant by $\#$ and $E' \vdash \varphi <: \mathtt{M}^-\ \tau$.

- In the preceding column, we must show that $E' \vdash \#\,\varphi <: \mathtt{R}\ \tau'$ (5). Here and since $E' \vdash \varphi <: \varphi'$, φ is one of the form $\mathtt{R}\ \tau$ or $\mathtt{V}\ \tau$, types τ and τ' are equal, and $\#\,\varphi$ is $\mathtt{R}\ \tau$. Hence (5).

Other subcases: given that $E' \vdash \varphi <: \varphi'$, the only remaining subcase is when φ is $\mathtt{M}\ \tau$ and φ' is $\mathtt{V}\ \tau'$. Then τ and τ' are equal and thus so are $\hat\varphi$ and φ'. Hence (1) and (3). Clearly, we also have $E' \vdash \mathtt{R}\ \tau <: \mathtt{M}\ \tau$ (2) and $E' \vdash \mathtt{M}\ \tau <: \mathtt{V}\ \tau$ (4).

Case (φ'', φ') is $(\mathtt{V}\ \tau_\ell, \mathtt{R}\ \tau_\ell)$, first line excluded: Then φ is either $\mathtt{M}\ \tau$ or $\mathtt{V}\ \tau$ with τ and τ_l equal and $\hat\varphi$ is $\mathtt{R}\ \tau$. Hence (1) and, since $\hat\varphi$ and φ' are here invariant by $\#$, we also have (3). Since both $E' \vdash \mathtt{R}\ \tau <: \mathtt{M}\ \tau$ and $E' \vdash \mathtt{R}\ \tau <: \mathtt{V}\ \tau$, we also have (2). The hypothesis $E' \vdash \varphi <: \mathtt{R}\ \tau_\ell$ never holds. However, $E' \vdash \mathtt{P}\ \tau_l <: \mathtt{R}\ \tau$ holds since τ and τ_l are equal.

Case (φ'', φ') is $(\mathtt{V}^-\ \tau_\ell, \mathtt{M}^-\ \tau_\ell)$, first line excluded: If φ is either $\mathtt{M}\ \tau$ or $\mathtt{V}\ \tau$ with τ and τ_l equal and we reason as in the previous case. That is τ and τ_l are equal and $\hat\varphi$ is $\mathtt{R}\ \tau$. In particular, (2) is unchanged. Since $E' \vdash \mathtt{R}\ \tau_l <: \mathtt{M}^-\ \tau_l$, we also have (1), (3) and $E' \vdash \mathtt{P}\ \tau_l <: \hat\varphi$. The hypothesis $E' \vdash \varphi <: \mathtt{M}^-\ \tau_\ell$ holds when φ is $\mathtt{M}\ \tau$, and then (4) holds since $\hat\varphi$ is φ.

Otherwise, φ is either $\mathtt{M}^-\ \tau$ or $\mathtt{V}^-\ \tau$ with $E' \vdash \tau_\ell <: \tau$ and $\hat\varphi$ is $\mathtt{M}^-\ \tau$. Since $E' \vdash \mathtt{M}^-\ \tau <: \mathtt{M}^-\ \tau_\ell$, i.e. $E' \vdash \hat\varphi <: \varphi'$, we have (1). Since $\mathtt{M}^-$, i.e. both sides, are invariant by $\#$, we also have (3). Since both $\#\,\varphi$ is equal to $\hat\varphi$, which is invariant by $\#$, we also have (2). The inequality $E' \vdash \varphi <: \varphi'$ holds when φ is $\mathtt{M}^-\ \tau$, i.e. $\hat\varphi$, and then $E' \vdash \varphi <: \hat\varphi$ trivially holds. Otherwise, (4) holds taking τ for τ'_ℓ.

■

B Subject reduction

Theorem 1 (Subject Reduction) *Typings are preserved by reduction. If $E \vdash a : \tau_a$ and $a \longrightarrow a'$ then $E \vdash a' : \tau_a$.*

Proof: By induction on the size of a and cases on the reduction.

Case Red Select: The expression a is of the form $v.\ell_j$ where v is $\zeta(\chi, \tau)[\ell_i = \varsigma(x) a_i\ ^{i \in I}]$ and j is in I. It reduces to $a_j\{\tau/\chi\}\{v/x\}$. The derivation of $E \vdash a : \tau_a$ ends with a subsumption rule preceded by rule Expr Select. Thus, there exists types τ' and τ''_j such that

$$E \vdash v : \tau' \qquad E \vdash \tau' <: \zeta(\chi)[\ell_j : \mathtt{R}^+\ \tau''_j\ ; \mathtt{U}]\ (1) \qquad E \vdash \tau''_j\{\tau'/\chi\} <: \tau_a\ (2)$$

The derivation $E \vdash v : \tau'$ itself ends with a subsumption rule preceded by rule Expr Object. Thus, τ is of the form $\zeta(\chi)[\rho]$ and there exist $\tau_i\ ^{i \in I}$ such that

$$E, \chi <: \#\,\tau, x_i : \chi \vdash a_i : \tau_i,\ \ \forall i \in I\ (3) \qquad E, \chi <: \#\,\tau \vdash (\ell_i : \mathtt{P}\ \tau_i\ ^{i \in I} ; \mathtt{A}) <: \rho\ (4) \qquad E \vdash \tau <: \tau'\ (5)$$

By transitivity between (5) and (1), we have $E \vdash \tau <: \zeta(\chi)[\ell_j : \mathtt{R}^+\ \tau''_j\ ; U]$ (6). By structural subtyping (lemma 3), and transitivity with (4), we have, in particular, $E, \chi <: \mathtt{T} \vdash \mathtt{P}\ \tau_j <: \mathtt{R}^+\ \tau''_j$.

By structural subtyping (lemma 3), $E, \chi <: \mathtt{T} \vdash \tau_j <: \tau''_j$. Thus, by subsumption applied to (3), $E, \chi <: \#\,\tau, x_i : \chi \vdash a_i : \tau''_j$ (7). The judgment (6) also implies that $E \vdash \tau <: \zeta(\chi)[\mathtt{U}]$, and by lemma 5 we have $E \vdash \tau <: \#\,\tau$. Therefore, applying substitution (lemma 2) to (7), we have $E, x_i : \tau \vdash a_i\{\tau/\chi\} : \tau''_j\{\tau/\chi\}$. Since $E \vdash v : \tau$, by substitution again, we have $E \vdash a_i\{\tau/\chi\}\{v/x\} : \tau''_j\{\tau/\chi\}$ (8). Since τ''_j is covariant, it follows from (5) that $E \vdash \tau''_j\{\tau/\chi\} <: \tau''_j\{\tau'/\chi\}$. By transitivity with (2), $E \vdash \tau''_j\{\tau/\chi\} <: \tau_a$ (9). We conclude using subsumption applied to (8) with (9).

Case Red Extend: The expression a is of the form $v.\ell \Leftarrow \zeta(\chi, \tau')\varsigma(x_0) a_0$ where v is $\zeta(\chi, \tau)[\ell_i = \varsigma(x_i) a_i\ ^{i \in I}]$ and ℓ is not one of the ℓ_i's. It reduces to $\zeta(\chi, \hat\tau)[\ell = \varsigma(x_0) a_0\ ; \ell_i = \varsigma(x_i) a_i\ ^{i \in I}]$ where $\hat\tau$ is $\tau \Leftarrow \tau'$.

Let τ and τ' be $\zeta(\chi)[\rho]$ and $\zeta(\chi)[\rho']$ (this is not restrictive) and $\hat\rho$ be $\rho \Leftarrow \rho'$. A derivation of $v.\ell_j \Leftarrow \zeta(\chi, \tau')\varsigma(x) a$ ends with a subsumption rule preceded by rule Expr Extend. Thus, τ' verifies:

$$(\varphi_0, \rho'(\ell)) \in \{(\mathtt{A}, \mathtt{P}\ \tau_\ell), (\mathtt{V}\ \tau_\ell, \mathtt{R}\ \tau_\ell), (\mathtt{V}^-\ \tau_\ell, \mathtt{M}^-\ \tau_\ell)\}\ (1) \qquad E \vdash v : \zeta(\chi)[\ell : \varphi_0\ ; \rho']\ (2)$$

$$E, \chi <: \#\,\tau', x_0 : \chi \vdash a_0 : \tau_\ell\ (3) \qquad E \vdash \tau' <: \tau_a\ (4)$$

A derivation of (2) ends with subsumption preceded by rule EXPR OBJECT. Thus, τ verifies:

$$E, \chi <: \# \tau, x_i : \chi \vdash a_i : \tau_i,\ \forall i \in I\ (5) \qquad E, \chi <: \# \tau \vdash (\ell_i : \mathtt{P}\ \tau_i{}^{i \in I}; \mathtt{A}) <: \rho\ (6)$$

$$E \vdash \tau <: \zeta(\chi)[\ell{:}\,\varphi_0\ ;\ \rho']\ (7)$$

From (7), by structural subtyping, we have $E, \chi <: \mathtt{T} \vdash \rho <: (\ell : \varphi_0; \rho')$. Thus, $E, \chi <: \mathtt{T} \vdash \rho \setminus \ell <: \rho' \setminus \ell$. This, together with (1) meets the hypotheses of lemma 4. Therefore,

$$E \vdash \hat{\tau} <: \tau'\ (8) \qquad E \vdash \# \hat{\tau} <: \# \tau\ (9) \qquad E \vdash \# \hat{\tau} <: \# \tau'\ (10)$$

Moreover, for some τ'_ℓ:

$$E, \chi <: \# \tau \vdash \rho \setminus \ell <: \hat{\rho} \setminus \ell\ (11) \qquad E, \chi <: \# \tau \vdash \tau_\ell <: \hat{\tau}'_\ell\ (12) \qquad E, \chi <: \# \tau \vdash \mathtt{P}\ \tau'_\ell <: \hat{\rho}(\ell)\ (13)$$

Combining (6), (11), and (13) we have $E, \chi <: \# \tau \vdash (\ell : \mathtt{P}\ \tau'_\ell; \ell_i : \mathtt{P}\ \tau_i{}^{i \in I}; \mathtt{A}) <: \hat{\rho}$. By bound weakening, since (9), we get $E, \chi <: \# \hat{\tau} \vdash (\ell : \mathtt{P}\ \tau'_\ell; \ell_i : \mathtt{P}\ \tau_i{}^{i \in I}; \mathtt{A}) <: \hat{\rho}$ (14). Combining (3) with (12), we have $E, \chi <: \# \tau', x_0 : \chi \vdash a_0 : \tau'_\ell$ (15). By substitution lemma applied to (15) and (5), with (9) we have

$$E, \chi <: \# \hat{\tau}, x_0 : \chi \vdash a_0 : \tau'_\ell \qquad E, \chi <: \# \hat{\tau}, x_i : \chi \vdash a_i : \tau_i,\ \forall i \in I$$

Combining with (14), we have $E \vdash a' : \hat{\tau}$. By subsumption applied with (8) and (4), we finally have $E \vdash a' : \tau_a$.

Case Red Update: We reuse the same notations. The difference is that ℓ is now one the ℓ_j for j in I. The expression a is of the form $v.\ell \Leftarrow \zeta(\chi, \tau')\varsigma(x)a$ where v is $\zeta(\chi, \tau)[\ell_i = \varsigma(x_i)a_i{}^{i \in I}]$. It reduces to $\zeta(\chi, \hat{\tau})[\ell_j = \varsigma(x)a\ ;\ \ell_i = \varsigma(x_i)a_i{}^{i \in I - j}]$.

We distinguish two subcases according to form of the typing derivation for a.

Subcase Expr Extend: This case is similar to the case for extension. The only differences in the proof if that here, from (6), (9) and (13), we have $E, \chi <: \# \hat{\tau} \vdash (\ell : \mathtt{P}\ \tau_\ell; \ell_i : \mathtt{P}\ \tau_i{}^{i \in I - j}; \mathtt{A}) <: \hat{\rho}$ instead of (14).

Subcase Expr Update: A derivation of $v.\ell \Leftarrow \zeta(\chi, \tau')\varsigma(x_0)a_0$ ends with a subsumption rule preceded by rule EXPR UPDATE. Thus, τ' verifies:

$$E \vdash v : \tau'\ (1) \qquad E \vdash \tau' <: \zeta(\chi)[\ell{:}\,\mathtt{R}\ \tau_\ell\ ;\ \rho_0]\ (2) \qquad E, \chi <: \# \tau', x_0 : \chi \vdash a_0 : \tau_\ell\ (3) \qquad E \vdash \tau' <: \tau_a\ (4)$$

A derivation of (1) ends with subsumption preceded by rule OBJECT. Thus, τ verifies:

$$E, \chi <: \# \tau, x_i : \chi \vdash a_i : \tau_i,\ \forall i \in I\ (5) \qquad E, \chi <: \# \tau \vdash (\ell_i : \mathtt{P}\ \tau_i{}^{i \in I}; \mathtt{A}) <: \rho\ (6) \qquad E \vdash \tau <: \tau'\ (7)$$

From (7), by structural subtyping, we have $E, \chi <: \mathtt{T} \vdash \rho <: \rho'$, which enables to apply lemma 4; we get

$$E \vdash \hat{\tau} <: \tau'\ (8) \qquad E \vdash \# \hat{\tau} <: \# \tau (9) \qquad E \vdash \# \hat{\tau} <: \# \tau' (10) \qquad E, \chi <: \# \tau \vdash \rho <: \hat{\rho}\ (11)$$

Combining (11) with (6), we have $E, \chi <: \# \tau \vdash (\ell_i : \mathtt{P}\ \tau_i{}^{i \in I}; \mathtt{A}) <: \hat{\rho}$. By bound weakening, since (9), we have $E, \chi <: \# \hat{\tau} \vdash (\ell_i : \mathtt{P}\ \tau_i{}^{i \in I}; \mathtt{A}) <: \hat{\rho}$ (12). By structural subtyping (lemma 3) applied to (2), we have $E, \chi <: \mathtt{T} \vdash \rho'(\ell) <: \mathtt{R}\ \tau_\ell$. By bound weakening with (10), we have $E, \chi <: \# \hat{\tau} \vdash \rho'(\ell) <: \mathtt{R}\ \tau_\ell$. By transitivity with (8) after applying structural subtyping, we get $E, \chi <: \# \hat{\tau} \vdash \hat{\rho}(\ell) <: \mathtt{R}\ \tau_\ell$. By structural subtyping, we must have $E, \chi <: \# \hat{\tau} \vdash \mathtt{P}\ \tau_\ell <: \hat{\rho}(\ell)$. Combining this with (12), we have $E, \chi <: \# \hat{\tau} \vdash (\ell : \mathtt{P}\ \tau_\ell; \ell_i : \mathtt{P}\ \tau_i{}^{i \in I - j}; \mathtt{A}) <: \hat{\rho}$ (13).

By substitution lemma applied to (3) and (5), we have

$$E, \chi <: \# \hat{\tau}, x : \chi \vdash a : \tau_\ell \qquad E, \chi <: \# \hat{\tau}, x_i : \chi \vdash a_i : \tau_i,\ \forall i \in I$$

Combining with (13), we have $E \vdash a' : \hat{\tau}$. By subsumption applied with (8) and (4), we finally have $E \vdash a' : \tau_a$.

Case Context: Trivial using the induction hypothesis.