Combining subsumption and binary methods: An object calculus with views

Jérôme Vouillon
Department of Computer and Information Science
University of Pennsylvania
vouillon@saul.cis.upenn.edu

March 15, 2000

Abstract

We present an object-oriented calculus which allows arbitrary hiding of methods in prototypes, even in the presence of binary methods and friend functions. This combination of features permits complete control of the interface a class exposes to the remainder of a program (which is of key importance for program readibility, security and ease of maintenance), while still allowing complex interactions with other classes belonging to a same module or software component. We also present a class-based extension of it.

This result is made possible by the use of *views*. A view is a name that specifies an interface to an object. A set of views is attached to each object and a method can be invoked either directly or via a view of the object.

Introduction

Information hiding is of key importance for making programs more readable, secure and maintenable. In particular, for abstraction purpose, one would expect to be able to specify freely what methods of a class are exported to the remainder of a program. Furthermore, abstraction should not be impeded by the need for complex interaction between objects. For instance, if objects from two different classes are to interact with one other, it should still be possible to hide the methods involved in the interaction. This means that the objects must be able to communicate via methods not present in the interfaces exported by the classes to the rest of the program. (A class or a function having such a privileged access to another class is commonly named a *friend*).

A structural subtyping setting, where methods are accessed by name and method names are global, does not provide such freedom [FR00]. Indeed, if a method is necessary for the interaction between two classes, it must remain in the interface of the class it belongs to (even though its access may be prevented using existential types [BF98]).

On the other hand, this problem does not exist in languages such as C++ or Java, where an explicitly declared inheritance hierarchy defines the subtyping relation between objects. The reason is that in this setting a method is selected according to both its name and the static type of the object. Then, a new method in a class will not conflict with a hidden method of the same name in a superclass. Furthermore, it is also possible to expose to the remainder of the program the fact that an object belongs to a subclass of a given class without having to export any of the methods of this class. Then, a friend function is simply a function which has access to some non-exported methods of a class. Type safety is achieved by ensuring that the function is only applied to objects of this class (or one of its subclasses).

However, such languages usually do not support some advanced features often found in languages using structural subtyping, such as selftype and binary methods [BCC+95]. We present in this paper a calculus combining the best of both worlds: it provides these two features while still allowing arbitrary method hiding and a proper encoding of friend functions. Rather than having classes, it uses more atomic constructions: it is prototype-based, and the notion that an object belongs to a given class is expressed by the fact that it possesses a given *view*. A view is simply a name (corresponding to a class name) giving access to an alternate interface to an object. So as to provide a clearer presentation and demonstrate that the calculus is of practical use, we also give a class-based extension to this calculus.

Several calculi dealing with similar issues have been proposed previously. Riecke and Stone [RS00] presented a calculus allowing arbitrary hiding of methods. This calculus only uses structural subtyping and does not have binary methods. More recently, Fisher and Reppy [FR00] proposed a calculus with support for inheritance-based subtyping. However, this calculus is first-order (that is, it does not have the notion of selftype). Furthermore, it does not allow a thorough encoding of friend functions: it cannot express the fact that a function has a privileged access to some methods of a class.

The paper is organized as follows: The first section provides some examples introducing our language. The core calculus is introduced in section 2. The full language and its translation into the calculus are presented in section 3. The encoding of friend functions is given in section 4. Type checking and type inference are discussed in section 5. Section 6 exposes some limitations of the calculus. Finally, related works are presented in section 7.

Notations

We write $f;(i = e)$ to denote the partial function that behaves exactly as f except for mapping i to e. This notation is not standard, but is very convenient, in particular for writing method lists. We write $f \subseteq g$ to denote that the function g extends the function f, that is, the graph of g is a superset of the graph of f.

1 Examples

We begin with a series of examples intended to give an idea of the full language. We then present some more complex examples made possible by the use of views.

1.1 Simple classes

As in Objective Caml [LVD+96], a class is similar to a function taking an argument (here x0) and returning an object, described by the class body. The class body contains the method definitions. The following class implements points. It has a single method get_x, which always returns the class parameter x0.

```
class simple_point (x0 : int) = object
  method get_x : int = x0
end
```

Objects of this class are created by the expression

```
new simple_point
```

of type `int -> Obj('a)[get_x: int]`: this is a function taking an integer as argument and returning an object. The type of the object lists the methods of the object. The type variable 'a stands for selftype and can occur in the method types. The method get_x of an object o of this class is invoked via the syntax o.get_x.

A view is a name that gives access to an alternative interface to an object: methods can be invoked or overridden either directly, or through a view via the syntax o.[v]m, where o is an object, v is a view of the object and m is a method of the view. The methods of self can only be invoked through a view. Indeed, in our calculus, the primary interface to self may be arbitrarily different in subclasses (for instance, some methods may have been hidden), but the set of views associated to self can only be augmented. Views are declared in the class declaration: in the example below, a view v is bound at the beginning of the class. This view is local to the class and provides a mean to access the methods of self. Self is explicitly bound at the beginning at the class body, together with selftype 'a. The method get_x is the same as in the previous example. The method set_x returns a copy of self, overriding the method get_x so that it returns x' rather than the class parameter x0: overriding is used here to model a modifiable object attribute.

```
class point [v] (x0 : int) = object (self : 'a)
  method get_x : int = x0
  method set_x (x' : int) : 'a =
    self.[v]get_x <- (fun self -> x')
end
```

A class can be extended by inheritance. The following class color_point has the same methods as the class point, plus a method get_color.

```
class color_point (c0 : string) (x0 : int) =
  object
    inherit point x0
    method get_color : string = c0
  end
```

The type of objects of this class is:

```
Obj('a)[get_x: int; set_x: int -> 'a; get_color: string]
```

This is a subtype of the type of an object of class point. As a consequence, a colored point can be considered as a point.

1.2 Binary methods

Let us now consider a class with a binary method, *i.e.* a method that takes an argument of the same type as the one of self. The class `comparable` has a method `x` that returns an integer, and a method `equal` that takes an argument `other` and checks whether the two invocations of the method `x` on respectively `other` and self return the same value.

```
class comparable [protected comp] (x0 : int) =
  object (self : 'a)
    method x : int = x0
    method equal (other : 'a) : bool =
      self.[comp]x = other.[comp]x
  end
```

It would not be correct to give the type `Obj('a)[x:int; equal:'a -> bool]` to objects of this class. Indeed, the argument of the method `equal` is expected to have a view `comp`. Therefore, this view must remain in the object type (it is placed at the end of the object type):

```
Obj('a)[x:int;equal:'a -> bool | comp]
```

This is achieved by introducing the view using the keyword `protected`. This keyword has two effects. First, the view is still defined after the class definition, but it is abstract (no method can be invoked via this view from outside the class body). Second, the view remains present in the type of an object of the class. As a consequence, the view behaves as a tag indicating that the object belongs to this class. (On the other hand, when no keyword is present, as in the previous example, the view is local to the class and does not occur in the type of instances of the class.)

We can define a subclass `comparable2` of the previous class that hides the method `x`, even though the binary method `equal` remains. This is possible because the method `equal` does not directly access the method `x` of its arguments, but accesses it via the view `comp`. The method `x` is hidden by restricting the signature of the class using a class type.

```
class comparable2 [protected comp2] (x0 : int) :
  object ('a)
    method equal : 'a -> bool
  end
= object inherit comparable x0 end
```

The type of the objects of this class is:

```
Obj('a)[equal:  'a -> bool | comp2]
```

Again, the view still occurs in this type. This way, the method `equal` of an object of class `comparable2` can only be applied to an object that also has the view `comp2`. Such an object must be an instance of either class `comparable2` or one of its subclasses. In our calculus, views are always inherited (but the type of an object may show only some of its views). Therefore, the object also has the inherited view `comp`, as required by the definition of method `equal`.

1.3 Public views

It is sometimes convenient to let a view remain concrete beyond the class definition where it is introduced. A typical example is the definition of *friend functions*, that is functions that have a privileged access to objects of a given class (see section 4 for a detailed example): such a function would access the object using the view rather than directly. The keyword `public` is used for this purpose.

```
class color_point2 [public cpoint] (c0 : string) (x0 : int) =
  object
    inherit point x0
    method get_color : string = c0
  end
```

The type of an object of class `color_point2` is the same as the type of an object of class `color_point`, except for the presence of the view `cpoint`.

```
Obj('a)[get_color: string; get_x: int; set_x: int -> 'a | cpoint]
```

As in class `comparable`, the view is present. However, while the view `comp` of the class `comparable` was abstract outside the class definition and could not be used to invoke methods, the view `cpoint` is concrete, and has type:

```
Obj('a)[get_color: string; get_x: int;
        set_x: int -> 'a]
```

Thus, one can create an object of class `color_point2` and invoke the method `get_color` either directly or via the view `cpoint`:

```
(new color_point2 "blue" 3).[cpoint]get_color
```

It is possible to write a subclass of class `color_point2` that hides the method `get_color` while retaining the view `cpoint`. Then, this method cannot be invoked directly, but it remains possible to invoke it via the view `cpoint`.

2 Core calculus

As in [FM95], the static semantics distinguishes two kinds of objects by their types: *prototypes* and *"proper objects"*. A prototype can be extended with new methods but it only supports trivial subtyping: hiding a method in a prototype is an explicit operation. On the other hand, a proper object is subject to implicit width subtyping, but it is not extensible. A prototype is turned into a proper object via subtyping.

The calculus has a notion of abstract methods. Abstract methods can only occur in prototypes. They are methods that are not yet defined but can be referenced from the definition of other methods. They will have to be defined prior to turning the prototype into a proper object. This notion is not mandatory. However, in order to ensure the soundness of a calculus with method hiding and object extension, method overriding and object extension must be distinguished. Indeed, in the first case, already existing methods can have access to the new definition of the method, while in the second case, the behavior of existing methods should not be altered. Rather than having these two operation as primitives as in [RS00], we have preferred to decompose object extension into the addition of an abstract method followed by the overriding of this method. We feel that these latter primitives are more atomic and orthogonal. Furthermore, they allow the definition of mutually recursive methods.

2.1 Syntax

The syntax of the calculus is presented in figure 1.

A view is a name that can be used to access a collection of methods of an object. It is explicitly introduced: the construction $\rho(k:t)a$ defines a new view k of type t with scope the expression a. This construction is similar to the one used in [WF91] to represent memory locations or to the ν binder of the π-calculus [MPW92]. The type $\zeta(\alpha)[\mathcal{L}]$ of a view is the type of the methods it gives access to, a finite partial function from method names to method types, parameterized by selftype α.

An expression $\zeta(x:\alpha)[L:\mathcal{L}]^{\varphi}_{\Phi}$ represents an object. The variable x stands for self, and the type variable α stands for selftype. The variable x is dynamically bound to the object upon method invocation. A method can be accessed either directly (via the *primary interface* of the object), or via a view (which provides an *alternative interface* to the object). In both cases, the external name of the method is translated into its internal name using a dictionary φ (a dictionary is a finite partial function from method names to names). As a consequence, an object holds a dictionary φ for its primary interface and a finite partial function Φ from views to dictionaries for its alternative interfaces. The finite partial function L maps internal method names to their definition, which are expressions. The types of the methods are given by the finite partial function $\mathcal{L}$ mapping internal method names to types.

The type $t_{\mathcal{K}}$ of a proper object is composed of the type of its primary interface t, and of the set $\mathcal{K}$ of its views. The type $t^{\mathcal{A}}_{\mathcal{K}}$ of a prototype also holds a set of method names $\mathcal{A}$. This set enumerates the abstract methods of the object.

The construction $a+(l:\varsigma(\alpha)\tau)$ allows to add an abstract method l of type τ to the object a, assuming that α is selftype. Symmetrically, the construction $a\setminus l$ is used to hide a method l. These constructions only affect the primary interface of the object. Indeed, similar constructions for the other interfaces would not make sense, as they have fixed types (which are the types of the corresponding views).

Methods can be invoked either directly ($a.l$) or via a specific view ($a._k l$). The two alternatives also exist for method overriding: $a.l \Leftarrow \varsigma(x:\alpha)a'$ and $a._k l \Leftarrow \varsigma(x:\alpha)a'$. It is possible to override a method only if it already occurs in the interface of the object. Method extension is achieved by first adding an abstract method and then overriding this method.

A view k can be added to an object a by the construction $\langle a\rangle_k$ if its type is the same as the type of the primary interface of the object. It is also possible to replace the primary interface of an object by the interface provided by one of its views k using the construction $a|_k$.

The standard packing and unpacking operations [CW85] provide view abstraction. The expression

$$\texttt{pack}\ a\ \texttt{as}\ \exists(k)\tau\ \texttt{hiding}\ \mathcal{K}$$

hides a set of view $\mathcal{K}$ using the abstract view k.

$a ::= x$		Variable
	$\mid \lambda(x:\tau)a$	Abstraction
	$\mid a(a')$	Application
	$\mid \zeta(x:\alpha)[L:\mathcal{L}]^{\varphi}_{\Phi}$	Object
	$\mid a + (l:\varsigma(\alpha)\tau)$	Method addition
	$\mid a \setminus l$	Method hiding
	$\mid a.l$	Method selection
	$\mid a._k l$	Method selection in view
	$\mid a.l \Leftarrow \varsigma(x:\alpha)a'$	Method override
	$\mid a._k l \Leftarrow \varsigma(x:\alpha)a'$	Method override in view
	$\mid \rho(k:t)a$	View binding
	$\mid \langle a \rangle_k$	View addition
	$\mid a\|_k$	View replacement
	$\mid \texttt{pack}\ a\ \texttt{as}\ \tau\ \texttt{hiding}\ \mathcal{K}$	Packing
	$\mid \texttt{open}\ a\ \texttt{as}\ [k,x]\ \texttt{in}\ a'$	Unpacking
$\tau ::= \alpha$		Type variable
	$\mid \tau \to \tau'$	Function type
	$\mid t_{\mathcal{K}}$	Object type
	$\mid t^{\mathcal{A}}_{\mathcal{K}}$	Prototype type
	$\mid \exists(k)\tau$	View abstraction
$t ::= \zeta(\alpha)[\mathcal{L}]$		View type

Figure 1: Syntax of the core calculus

2.2 Dynamic semantics

We give a small-step reduction semantics to our calculus. The local reduction relation $\longrightarrow$ is defined in figure 2. The rules defining this relation make use of several substitution operations written $a\{v/x\}$, $a\{\tau/\alpha\}$ and $a\{\mathcal{K}/k\}$. The first two are standard value and type substitution. The third one substitutes a set of views for a single one. It is such that:

$$\begin{array}{rcl} \mathcal{K}\{\mathcal{K}'/k\} & = & \mathcal{K} \setminus \{k\} \cup \mathcal{K}' \text{ if } k \in \mathcal{K} \\ & = & \mathcal{K} \text{ otherwise.} \end{array}$$

and is otherwise similar to the other substitutions.

Method addition modifies the internal type $\mathcal{L}$ of the object and the object primary dictionary φ: a fresh method l' is added to the type $\mathcal{L}$ and the dictionary is extended in order to map the external method name l to the internal method name l'. Method hiding only affects the primary dictionary: hiding the method l consists in removing the name of the method from the dictionary, so that the method cannot be accessed via this dictionary anymore. Overriding consists in defining or redefining a method, by modifying the method list L. Overriding can be done either directly or via a view k. In the first case, the method name is translated using the primary dictionary φ. In the second case, the dictionary associated to the view k, that is $\Phi(k)$, is used. For the invocation of a method l, the definition of this method is retrieved from L. In this definition, selftype is replaced by a suitable type and self is replaced by the object. The capture of a view k consists in binding in Φ this view to the current primary dictionary φ. The selection of a view k consists in replacing the primary dictionary by the dictionary $\Phi(k)$ associated to the view. For unpacking an existentially abstracted expression, the abstracted view k is replaced by its actual value, that is the set of views $\mathcal{K}$, and the variable x is replaced by the packed value v'. The last reduction rule pushes a view binding outwards to toplevel.

The semantics is defined using the contexts E and F:

$$\begin{array}{ll} E ::= [\_] & F ::= [\_] \\ \quad \mid \rho(k:t)E & \quad \mid F + (l : \varsigma(\alpha)\tau) \\ & \quad \mid F.l \Leftarrow \varsigma(x:\alpha)a \\ & \quad \mid F._k l \Leftarrow \varsigma(x:\alpha)a \\ & \quad \mid F.l \\ & \quad \mid F._k l \\ & \quad \mid \langle F \rangle_k \\ & \quad \mid F|_k \\ & \quad \mid F(a) \\ & \quad \mid v(F) \\ & \quad \mid \texttt{pack } F \texttt{ as } \tau \texttt{ hiding } \mathcal{K} \\ & \quad \mid \texttt{open } F \texttt{ as } [k,x] \texttt{ in } a \end{array}$$

Evaluations context are context of the form $E[F[\_]]$: the local reduction relation $\longrightarrow$ is extended to a one-step evaluation relation: $a \longrightarrow a'$ iff there are expressions a_1 and a_1' such that $a = E[F[a_1]]$, $a_1 \longrightarrow a_1'$ and $a' = E[F[a_1']]$.

Values are defined by the following sub-grammar of expressions:

$$\begin{array}{lr} v ::= x & \text{Variable} \\ \quad \mid \lambda(x:\tau)a & \text{Abstraction} \end{array}$$

Method addition ($l' \notin \operatorname{dom} \mathcal{L} \cup \operatorname{range} \varphi \cup \bigcup_{k \in \operatorname{dom} \Phi} \operatorname{range} \Phi(k)$)

$$\zeta(x:\alpha)[L:\mathcal{L}]^{\varphi}_{\Phi} + (l : \varsigma(\alpha)\tau) \quad \longrightarrow \quad \zeta(x:\alpha)[L:(\mathcal{L};(l':\tau))]^{\varphi;(l=l')}_{\Phi}$$

Method hiding

$$\zeta(x:\alpha)[L:\mathcal{L}]^{\varphi}_{\Phi} \setminus l \quad \longrightarrow \quad \zeta(x:\alpha)[L:\mathcal{L}]^{\varphi|_{\operatorname{dom}\varphi \setminus \{l\}}}_{\Phi}$$

Override

$$\zeta(x:\alpha)[L:\mathcal{L}]^{\varphi}_{\Phi}.l \Leftarrow \varsigma(x:\alpha)a \quad \longrightarrow \quad \zeta(x:\alpha)[L;(\varphi(l)=a):\mathcal{L}]^{\varphi}_{\Phi}$$
$$\zeta(x:\alpha)[L:\mathcal{L}]^{\varphi}_{\Phi}.{}_k l \Leftarrow \varsigma(x:\alpha)a \quad \longrightarrow \quad \zeta(x:\alpha)[L;(\Phi(k)(l)=a):\mathcal{L}]^{\varphi}_{\Phi}$$

Method invocation ($v = \zeta(x:\alpha)[L:\mathcal{L}]^{\varphi}_{\Phi}$ and $\tau = \zeta(\alpha)[\mathcal{L} \circ \varphi]_{\operatorname{dom}\Phi}$)

$$v.l \quad \longrightarrow \quad L(\varphi(l))\{\tau/\alpha\}\{v/x\}$$
$$v.{}_k l \quad \longrightarrow \quad L(\Phi(k)(l))\{\tau/\alpha\}\{v/x\}$$

View manipulations

$$\langle \zeta(x:\alpha)[L:\mathcal{L}]^{\varphi}_{\Phi} \rangle_k \quad \longrightarrow \quad \zeta(x:\alpha)[L:\mathcal{L}]^{\varphi}_{\Phi;(k=\varphi)}$$
$$\zeta(x:\alpha)[L:\mathcal{L}]^{\varphi}_{\Phi} \,|_k \quad \longrightarrow \quad \zeta(x:\alpha)[L:\mathcal{L}]^{\Phi(k)}_{\Phi}$$

Application

$$(\lambda(x:\tau)a)(v) \quad \longrightarrow \quad a\{v/x\}$$

Unpacking ($v = \texttt{pack}\ v'\ \texttt{as}\ \exists(k)\tau\ \texttt{hiding}\ \mathcal{K}$)

$$\texttt{open}\ v\ \texttt{as}\ [k,x]\ \texttt{in}\ a \quad \longrightarrow \quad a\{\mathcal{K}/k\}\{v'/x\}$$

View bindings ($F \neq [\_]$)

$$F[\rho(k:t)a] \quad \longrightarrow \quad \rho(k:t)F[a]$$

Figure 2: Reduction rules

$\zeta(x : \alpha)[L : \mathcal{L}]^{\varphi}_{\Phi}$	Object
$\mid$ `pack` v `as` τ `hiding` $\mathcal{K}$	View abstraction

The semantics we give is not deterministic. Indeed, when a method is added to a prototype, an internal method name can be arbitrarily chosen for this method. It would be easy to make this semantics deterministic by providing a choice function, associating a new method names to the set of already existing internal method names (this is what is done in [RS00], where the methods are numbered). Alternatively, one can notice that internal method names cannot be observed from outside an object, in the sense that the behavior of the object is the same whatever method name is chosen. It would therefore be possible to identify objects modulo renaming of their internal method names.

2.3 Type system

An environment contains both value bindings and view bindings. It also contains what we call *matching hypothesis*, that is the hypothesis that a type α has at least the set of views $\mathcal{K}$ (this is not the same notion of matching as the one introduced by Kim Bruce [BFP97], but the two notions have similarities: types that match a given set of views have a similar shape, but are not necessarily subtypes of a same type).

$\Gamma ::= \emptyset$	Empty environment
$\mid \Gamma; (x : \tau)$	Value binding
$\mid \Gamma; (\alpha <\# \mathcal{K})$	Matching hypothesis
$\mid \Gamma; (k : t)$	View binding
$\mid \Gamma; (k)$	Abstract view binding

We make use of three kinds of judgments : typing judgments $\Gamma \vdash a : \tau$, subtyping judgments $\Gamma \vdash \tau \preceq \tau'$ and matching judgments $\Gamma \vdash \alpha <\# \mathcal{K}$.

A type τ, a view type t, or an expression a are *closed* with respect to an environment Γ if all of their variables are bound in this environment. An environment Γ is closed if all of the types and view types of its range are closed with respect to Γ. A typing judgment $\Gamma \vdash a : \tau$ is closed if Γ is closed and a and τ are closed with respect to Γ; a subtyping judgment $\Gamma \vdash \tau \preceq \tau'$ is closed if Γ is closed and τ and τ' are closed with respect to Γ. We assume that all judgments under discussion are closed, and that no variables are ever bound twice in an environment.

Some typing rules requires that some type variable α appears covariantly in some type τ (which we note $\mathrm{co}_\alpha(\tau)$). As usual, we say that the type variable α appears *covariantly* in the type τ if any of the following is true:

- α is not free in τ;
- τ is α;
- τ is $\tau_1 \to \tau_2$ and α appears contravariantly in τ_1 and covariantly in τ_2.

Similarly, the type variable α appears *contravariantly* in the type τ if any of the following is true:

- α is not free in τ;
- τ is $\tau_1 \to \tau_2$ and α appears covariantly in τ_1 and contravariantly in τ_2.

The typing rules are given in figure 3, 4 and 5. We only describe some of the typing rules. The rule PROTO is the most complex one. It first defines the different components of the prototype's type. The type $\mathcal{L}'$ of the prototype's primary interface is the translation by the dictionary φ of the internal method types of the prototype. The abstract methods $\mathcal{A}$ are the methods from $\mathcal{L}'$ which are not defined, that is, that does not occur in the list of methods L of the prototype. The set of views $\mathcal{K}$ of the prototype is the domain of the function Φ. The condition $\operatorname{dom}\mathcal{L} \subseteq \operatorname{dom} L \cup \operatorname{range}\varphi$ means that the methods of $\mathcal{L}$ either are methods defined by the prototype, or are visible from outside of the prototype. The next condition means that $\mathcal{L}$ must provide a type to all methods of L. The important point here is that selftype does not depend on the type $\mathcal{L}'$ of the primary interface. Therefore, it will be possible to hide any method of this type freely. Finally, the last condition means that the views of the prototype must not be abstract, and that the type of a view k corresponds to the translation of $\mathcal{L}$ by the dictionary $\Phi(k)$ associated to this view.

The type of the expression $a + (l : \varsigma(\alpha)\tau)$ (rule EXTENSION) is the type of the expression a where the method l has been added to the primary interface $\mathcal{L}$ and the set $\mathcal{A}$ of abstract methods. The type of the expression $a \setminus l$ (rule RESTRICTION) is the type of the expression a where the method l has been removed from the primary interface $\mathcal{L}$, provided that this method is not abstract.

A view can be captured by an object provided that its type is the type of the primary interface of the object (rule VIEW-CAPTURE). The view is added to the set of views of the object type. One can invoke a method l via a view k (rule VIEW-SELECTION) provided that this view is one of the views of the object and that it is not abstract. The type of the result of the invocation is the type of the method in the view, where the type of the object has been substituted for selftype.

2.4 Soundness

We have proved the following soundness results:

Theorem 1 (Subject reduction) *If $\Gamma \vdash a : \tau$ and $a \longrightarrow a'$, then $\Gamma \vdash a' : \tau$.*

Theorem 2 (Progress) *If $\vdash a : \tau$ then a is $E[v]$ for some value v or $a \longrightarrow a'$ for some expression a'.*

The proofs are given in appendix A.

3 Class language

3.1 Syntax

We present a simple extension of the core calculus with classes. The syntax of classes, presented in figure 6 is almost standard. We have however made

$$\textsc{(Var)}\quad \frac{(x:\tau)\in\Gamma}{\Gamma\vdash x:\tau}
\qquad\qquad
\textsc{(Abs)}\quad \frac{\Gamma;(x:\tau')\vdash a:\tau}{\Gamma\vdash \lambda(x:\tau')a:\tau'\to\tau}$$

$$\textsc{(App)}\quad \frac{\Gamma\vdash a:\tau'\to\tau \qquad \Gamma\vdash a':\tau'}{\Gamma\vdash a(a'):\tau}
\qquad\qquad
\textsc{(Sub)}\quad \frac{\Gamma\vdash a:\tau \qquad \Gamma\vdash \tau\preceq\tau'}{\Gamma\vdash a:\tau'}$$

$$\textsc{(Proto)}\quad \frac{\begin{array}{ll}
\mathcal{L}'=\mathcal{L}\circ\varphi & \mathcal{A}=\mathrm{dom}\,\mathcal{L}'\setminus\mathrm{dom}(L\circ\varphi)\\
\mathcal{K}=\mathrm{dom}\,\Phi & \mathrm{dom}\,\mathcal{L}\subseteq\mathrm{dom}\,L\cup\mathrm{range}\,\varphi\\
\multicolumn{2}{l}{\text{For all } l \text{ in } \mathrm{dom}\,L,\ \Gamma;(\alpha <\#\ \mathcal{K});(x:\alpha)\vdash L(l):\mathcal{L}(l)}\\
\multicolumn{2}{l}{\forall k\in\mathrm{dom}\,\Phi\cdot(k:\zeta(\alpha)[\mathcal{L}''])\in\Gamma\wedge\mathcal{L}''=\mathcal{L}\circ\Phi(k)}
\end{array}}{\Gamma\vdash\zeta(x:\alpha)[L:\mathcal{L}]^{\varphi}_{\Phi}:\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}}}$$

$$\textsc{(Extension)}\quad \frac{\Gamma\vdash a:\zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}}{\Gamma\vdash a+(l:\varsigma(\alpha)\tau):\zeta(\alpha)[\mathcal{L};(l:\tau)]^{\mathcal{A}\cup\{l\}}_{\mathcal{K}}}$$

$$\textsc{(Restriction)}\quad \frac{\Gamma\vdash a:\zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}} \qquad l\in\mathrm{dom}\,\mathcal{L}\setminus\mathcal{A}}{\Gamma\vdash a\setminus l:\zeta(\alpha)[\mathcal{L}|_{\mathrm{dom}\,\mathcal{L}\setminus\{l\}}]^{\mathcal{A}}_{\mathcal{K}}}$$

$$\textsc{(Selection)}\quad \frac{\begin{array}{c}\Gamma\vdash a:\tau \qquad \tau=\zeta(\alpha)[\mathcal{L}]_{\mathcal{K}}\\ (l:\tau')\in\mathcal{L}\end{array}}{\Gamma\vdash a.l:\tau'\{\tau/\alpha\}}
\qquad\qquad
\textsc{(View-Selection)}\quad \frac{\begin{array}{c}\Gamma\vdash a:\tau \qquad \Gamma\vdash\tau <\#\ \mathcal{K}\\ k\in\mathcal{K} \qquad (k:\zeta(\alpha)[\mathcal{L}'])\in\Gamma\\ (l:\tau')\in\mathcal{L}'\end{array}}{\Gamma\vdash a._{k}l:\tau'\{\tau/\alpha\}}$$

$$\textsc{(Override)}\quad \frac{\begin{array}{c}\Gamma\vdash a:\zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}\\ \Gamma;(\alpha <\#\ \mathcal{K});(x:\alpha)\vdash a':\mathcal{L}(l)\end{array}}{\Gamma\vdash a.l \Leftarrow \varsigma(x:\alpha)a':\zeta(\alpha)[\mathcal{L}]^{\mathcal{A}\setminus\{l\}}_{\mathcal{K}}}
\qquad\qquad
\textsc{(View-Override)}\quad \frac{\begin{array}{c}\Gamma\vdash a:\tau \qquad \Gamma\vdash\tau <\#\ \mathcal{K}\\ k\in\mathcal{K} \qquad (k:\zeta(\alpha)[\mathcal{L}'])\in\Gamma\\ \Gamma;(\alpha <\#\ \mathcal{K});(x:\alpha)\vdash a':\mathcal{L}'(l)\end{array}}{\Gamma\vdash a._{k}l \Leftarrow \varsigma(x:\alpha)a':\tau}$$

$$\textsc{(View-Abs)}\quad \frac{\Gamma;(k:t)\vdash a:\tau}{\Gamma\vdash\rho(k:t)a:\tau}
\qquad
\textsc{(View-Capture)}\quad \frac{\begin{array}{c}\Gamma\vdash a:\zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}\\ (k:\zeta(\alpha)[\mathcal{L}])\in\Gamma\end{array}}{\Gamma\vdash\langle a\rangle_{k}:\zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}\cup\{k\}}}
\qquad
\textsc{(View-Replace)}\quad \frac{\begin{array}{c}\Gamma\vdash a:\zeta(\alpha)[\mathcal{L}]_{\mathcal{K}}\\ (k:\zeta(\alpha)[\mathcal{L}'])\in\Gamma\\ k\in\mathcal{K}\end{array}}{\Gamma\vdash a|_{k}:\zeta(\alpha)[\mathcal{L}']_{\mathcal{K}}}$$

$$\textsc{(Pack)}\quad \frac{\Gamma\vdash a:\tau\{\mathcal{K}/k\}}{\Gamma\vdash \texttt{pack}\ a\ \texttt{as}\ \exists(k)\tau\ \texttt{hiding}\ \mathcal{K}:\exists(k)\tau}
\qquad\qquad
\textsc{(Open)}\quad \frac{\begin{array}{c}\Gamma\vdash a':\exists(k)\tau'\\ \Gamma;(k);(x:\tau')\vdash a:\tau\end{array}}{\Gamma\vdash \texttt{open}\ a'\ \texttt{as}\ [k,x]\ \texttt{in}\ a:\tau}$$

Figure 3: Typing rules

$$\text{(Sub-Refl)}\quad \frac{}{\Gamma \vdash \tau \preceq \tau}$$

$$\text{(Sub-Arrow)}\quad \frac{\Gamma \vdash \tau_1 \preceq \tau_2 \qquad \Gamma \vdash \tau_2' \preceq \tau_1'}{\Gamma \vdash \tau_1' \to \tau_1 \preceq \tau_2' \to \tau_2}$$

$$\text{(Sub-Proto)}\quad \frac{\Gamma \vdash \zeta(\alpha)[\mathcal{L}]_{\mathcal{K}} \preceq \zeta(\alpha)[\mathcal{L}']_{\mathcal{K}'}}{\Gamma \vdash \zeta(\alpha)[\mathcal{L}]^{\emptyset}_{\mathcal{K}} \preceq \zeta(\alpha)[\mathcal{L}']_{\mathcal{K}'}}$$

$$\text{(Sub-Object)}\quad \frac{\begin{array}{c}\mathcal{L}' \subseteq \mathcal{L} \qquad \mathrm{co}_\alpha(\mathcal{L}') \qquad \mathcal{K}' \subseteq \mathcal{K} \\ \forall k \in \mathcal{K}' \cdot (k : \zeta(\alpha)[\mathcal{L}'']) \in \Gamma \wedge \mathrm{co}_\alpha(\mathcal{L}'')\end{array}}{\Gamma \vdash \zeta(\alpha)[\mathcal{L}]_{\mathcal{K}} \preceq \zeta(\alpha)[\mathcal{L}']_{\mathcal{K}'}}$$

$$\text{(Sub-Match)}\quad \frac{\begin{array}{c}\Gamma \vdash \alpha <\# \mathcal{K} \\ \forall k \in \mathcal{K} \cdot (k : \zeta(\alpha)[\mathcal{L}'']) \in \Gamma \wedge \mathrm{co}_\alpha(\mathcal{L}'')\end{array}}{\Gamma \vdash \alpha \preceq \zeta(\alpha')[\emptyset]_{\mathcal{K}}}$$

Figure 4: Subtyping rules

$$\text{(Match-Var)}\quad \frac{(\alpha <\# \mathcal{K}) \in \Gamma \qquad \mathcal{K}' \subseteq \mathcal{K}}{\Gamma \vdash \alpha <\# \mathcal{K}'}$$

$$\text{(Match-Obj)}\quad \frac{\mathcal{K}' \subseteq \mathcal{K}}{\Gamma \vdash \zeta(\alpha)[\mathcal{L}]_{\mathcal{K}} <\# \mathcal{K}'}$$

Figure 5: Matching rules

$a ::=$	$\ldots$	
	$\mid$ `class` $c\,[k](x:\tau) = \mathit{expr}$ `in` a	Private class
	$\mid$ `class` $c\,[$`protected` $k](x:\tau) = \mathit{expr}$ `in` a	Protected class
	$\mid$ `class` $c\,[$`public` $k](x:\tau) = \mathit{expr}$ `in` a	Public class
	$\mid$ `new` c	Object creator
$\mathit{expr} ::=$	$(\mathit{expr} : \mathit{type})$	Class constraint
	$\mid$ `object` body `end`	Class body
$\mathit{body} ::=$	$\mathit{extension}$	
	$\mid$ body `method` $l = \varsigma(x:\alpha)a$	Method definition
$\mathit{extension} ::=$	$\mathit{inheritance}$	
	$\mid$ $\mathit{extension}$ `abstract` $l : \varsigma(\alpha)\tau$	Method addition
$\mathit{inheritance} ::=$	$\emptyset$	
	$\mid$ `inherit` $c(a)$	Inheritance
$\mathit{type} ::=$	`object` $\mathit{ty\text{-}body}$ `end`	Class type
$\mathit{ty\text{-}body} ::=$	$\mathit{ty\text{-}inherit}$	
	$\mid$ $\mathit{ty\text{-}body}$ `abstract` $l : \varsigma(\alpha)\tau$	Abstract method
	$\mid$ $\mathit{ty\text{-}body}$ `method` $l : \varsigma(\alpha)\tau$	Concrete method
$\mathit{ty\text{-}inherit} ::=$	$\emptyset$	
	$\mid$ `inherit` c	Class type inheritance

Figure 6: Classes syntax

some restriction so that the translation of this language into the core calculus be simpler. First, the allocation of a slot for a method is distinct from the definition or overriding of a method, and second, the components of a class body are ordered : first the inheritance clause, then the allocation of methods, and finally the definition of methods.

For the sake of conciseness, the examples at the beginning of the paper make use of some syntactic sugar: method definition and method addition are combined into a single construction; self and selftype are bound only once rather than at each method definition.

3.2 Translation

We now define a typed translation of the full language into the core calculus. Environment are extended with class bindings:

$$\Gamma ::= \ldots$$

$$\frac{}{\Gamma \vdash \emptyset : \zeta(\alpha)[\emptyset]^{\emptyset}_{\emptyset}}\ (\textsc{Type-Empty}) \qquad \frac{(c : (\mathcal{K}', \tau' \to \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}})) \in \Gamma}{\Gamma \vdash \texttt{inherit}\ c : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}\setminus\mathcal{K}'}}\ (\textsc{Type-Inher})$$

$$\frac{\Gamma \vdash \mathit{body} : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}}{\Gamma \vdash (\mathit{body}\ \texttt{abstract}\ l : \varsigma(\alpha)\tau) : \zeta(\alpha)[\mathcal{L}; (l : \tau)]^{\mathcal{A}\cup\{l\}}_{\mathcal{K}}}\ (\textsc{Abstract-Method})$$

$$\frac{\Gamma \vdash \mathit{body} : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}}{\Gamma \vdash (\mathit{body}\ \texttt{method}\ l : \varsigma(\alpha)\tau) : \zeta(\alpha)[\mathcal{L}; (l : \tau)]^{\mathcal{A}\setminus\{l\}}_{\mathcal{K}}}\ (\textsc{Concrete-Method})$$

$$\frac{\Gamma \vdash \textit{ty-body} : \tau}{\Gamma \vdash \texttt{object}\ \textit{ty-body}\ \texttt{end} : \tau}\ (\textsc{Class-Type})$$

Figure 7: Class type translation

$$\mid \Gamma; (c : (\mathcal{K}', \tau \to \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}})) \qquad \text{Class binding}$$

The type of a class is a pair $(\mathcal{K}', \tau \to \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}})$. The type τ is the type of the class argument, while $\zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}$ represents the type of the body of the class. The set $\mathcal{K}'$ contains the hidden views of the class. Indeed, the static semantics of the core calculus does not allow to hide all views of a prototype: it is only possible to coalesce a set of views as one abstract view. The set $\mathcal{K}'$ is thus used to track the views that should be hidden in the type of an object of the class (which is therefore $\zeta(\alpha)[\mathcal{L}]_{\mathcal{K}\setminus\mathcal{K}'}$).

Translations rules are given in figures 7, 8 and 9. They define the following judgments:

- $\Gamma \vdash \mathit{type} : \tau$, $\Gamma \vdash \textit{ty-body} : \tau$ and $\Gamma \vdash \textit{ty-inherit} : \tau$ for class types;
- $\Gamma \vdash \mathit{expr} : (\mathcal{K}, \tau) \Rightarrow \overline{a}$, $\Gamma \vdash \mathit{body} : (\mathcal{K}, \tau) \Rightarrow \overline{a}$, $\Gamma \vdash \mathit{extension} : (\mathcal{K}, \tau) \Rightarrow \overline{a}$ and $\Gamma \vdash \mathit{inheritance} : (\mathcal{K}, \tau) \Rightarrow \overline{a}$ for class expressions;
- $\Gamma \vdash a : \tau \Rightarrow \overline{a}$ for other expressions.

The translation rules for the core calculus expressions have been omitted, as they are obvious: for instance, here is the one corresponding to method overriding:

$$\frac{\Gamma \vdash a : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}} \Rightarrow \overline{a} \qquad \Gamma; (\alpha <\!\# \mathcal{K}); (x : \alpha) \vdash a' : \mathcal{L}(l) \Rightarrow \overline{a'}}{\Gamma \vdash a.l \Leftarrow \varsigma(x : \alpha)a' : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}\setminus\{l\}}_{\mathcal{K}} \Rightarrow \overline{a}.l \Leftarrow \varsigma(x : \alpha)\overline{a'}}$$

The translation of class expressions is simple, as for each high-level construction there is a directly corresponding construction in the core calculus. Class bindings are more complicated, and we therefore describe the rules Private-Class

(INHERITANCE)

$$\frac{(c : (\mathcal{K}, \tau' \to \tau)) \in \Gamma \qquad \Gamma \vdash a : \tau_1' \Rightarrow \overline{a} \qquad \Gamma \vdash \tau_1' \preceq \tau'}{\Gamma \vdash \texttt{inherit}\ c(a) : (\mathcal{K}, \tau) \Rightarrow c(\overline{a})}$$

(EMPTY)

$$\frac{}{\Gamma \vdash \emptyset : (\emptyset, \zeta(\alpha)[\emptyset]^{\emptyset}_{\emptyset}) \Rightarrow \zeta(x : \alpha)[\emptyset : \emptyset]^{\emptyset}_{\emptyset}}$$

(ABSTRACT-METHOD)

$$\frac{\Gamma \vdash \mathit{extension} : (\mathcal{K}', \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}) \Rightarrow a}{\Gamma \vdash (\mathit{extension}\ \texttt{abstract}\ l : \varsigma(\alpha)\tau) : (\mathcal{K}', \zeta(\alpha)[\mathcal{L}; (l : \tau)]^{\mathcal{A} \cup \{l\}}_{\mathcal{K}}) \Rightarrow a + (l : \varsigma(\alpha)\tau)}$$

(LOCAL-VIEW)

$$\frac{\Gamma \vdash \mathit{extension} : (\mathcal{K}', \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}) \Rightarrow a}{\Gamma, (k : \zeta(\alpha)[\mathcal{L}]) \vdash \mathit{extension} : (\mathcal{K}' \cup \{k\}, \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K} \cup \{k\}}) \Rightarrow \langle a \rangle_k}$$

(METHOD)

$$\frac{\Gamma, (k : \zeta(\alpha)[\mathcal{L}]) \vdash \mathit{body} : (\mathcal{K}', \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}) \Rightarrow a' \qquad \Gamma; (k : \zeta(\alpha)[\mathcal{L}]); (\alpha <\# \mathcal{K}); (x : \alpha) \vdash a : \mathcal{L}(l) \Rightarrow \overline{a}}{\Gamma, (k : \zeta(\alpha)[\mathcal{L}]) \vdash \mathit{body}\ \texttt{method}\ l = \varsigma(x : \alpha)a : (\mathcal{K}', \zeta(\alpha)[\mathcal{L}]^{\mathcal{A} \setminus \{l\}}_{\mathcal{K}}) \Rightarrow a'.l \Leftarrow \varsigma(x : \alpha)\overline{a}}$$

(CLASS-BODY)

$$\frac{\Gamma, (k : \zeta(\alpha)[\mathcal{L}]) \vdash \mathit{body} : (\mathcal{K}, \tau) \Rightarrow a}{\Gamma, (k : \zeta(\alpha)[\mathcal{L}]) \vdash \texttt{object}\ \mathit{body}\ \texttt{end} : (\mathcal{K}, \tau)\tau \Rightarrow a}$$

(CLASS-CONSTRAINT)

$$\frac{\Gamma, (k : \zeta(\alpha)[\mathcal{L}]) \vdash \mathit{expr} : (\mathcal{K}', \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}) \Rightarrow a \qquad \Gamma \vdash \mathit{type} : \zeta(\alpha)[\mathcal{L}_1]^{\mathcal{A}}_{\mathcal{K}_1} \qquad \mathcal{L}_1 \subseteq \mathcal{L} \qquad \mathcal{K}_1 \subseteq \mathcal{K} \qquad \mathcal{A} \subseteq \operatorname{dom} \mathcal{L}_1}{\Gamma, (k : \zeta(\alpha)[\mathcal{L}]) \vdash (\mathit{expr} : \mathit{type}) : (\mathcal{K} \setminus \mathcal{K}_1, \zeta(\alpha)[\mathcal{L}_1]^{\mathcal{A}}_{\mathcal{K}}) \Rightarrow a \setminus (\operatorname{dom} \mathcal{L} \setminus \operatorname{dom} \mathcal{L}_1)}$$

Figure 8: Class translation

(Private-Class)

$$\frac{\begin{array}{c}\Gamma;(x:\tau'),(k:\zeta(\alpha)[\mathcal{L}])\vdash \mathit{expr}:(\mathcal{K}',\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}})\Rightarrow a'\\ \Gamma;(k);(c:(\{k\},\tau'\to\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\setminus\mathcal{K}'\cup\{k\}}))\vdash a:\tau\Rightarrow\overline{a}\end{array}}{\begin{array}{c}\Gamma\vdash \texttt{class}\ c\ [k](x:\tau')=\mathit{expr}\ \texttt{in}\ a:\tau\Rightarrow\\ \texttt{open}\ (\rho(k:\zeta(\alpha)[\mathcal{L}])\texttt{pack}\ \lambda(x:\tau')a'\ \texttt{as}\ \exists(k')(\tau'\to\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\setminus\mathcal{K}'\cup\{k'\}})\\ \texttt{hiding}\ \mathcal{K}')\ \texttt{as}\ [k,c]\ \texttt{in}\ \overline{a}\end{array}}$$

(Protected-Class)

$$\frac{\begin{array}{c}\Gamma;(x:\tau'),(k:\zeta(\alpha)[\mathcal{L}])\vdash \mathit{expr}:(\mathcal{K}',\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}})\Rightarrow a'\\ \Gamma;(k);(c:(\emptyset,\tau'\to\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\setminus\mathcal{K}'\cup\{k\}}))\vdash a:\tau\Rightarrow\overline{a}\end{array}}{\begin{array}{c}\Gamma\vdash \texttt{class}\ c\ [\texttt{protected}\ k](x:\tau')=\mathit{expr}\ \texttt{in}\ a:\tau\Rightarrow\\ \texttt{open}\ (\rho(k:\zeta(\alpha)[\mathcal{L}])\texttt{pack}\ \lambda(x:\tau')a'\ \texttt{as}\ \exists(k')(\tau'\to\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\setminus\mathcal{K}'\cup\{k'\}})\\ \texttt{hiding}\ \mathcal{K}')\ \texttt{as}\ [k,c]\ \texttt{in}\ \overline{a}\end{array}}$$

(Public-Class)

$$\frac{\begin{array}{c}\Gamma;(x:\tau'),(k:\zeta(\alpha)[\mathcal{L}])\vdash \mathit{expr}:(\mathcal{K}',\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}})\Rightarrow a'\\ \Gamma;(k'');(k':\zeta(\alpha)[\mathcal{L}']);(c:(\emptyset,\tau'\to\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\setminus\mathcal{K}'\cup\{k',k''\}}))\vdash a\{k'/k\}:\tau\Rightarrow\overline{a}\end{array}}{\begin{array}{c}\Gamma\vdash \texttt{class}\ c\ [\texttt{public}\ k](x:\tau')=\mathit{expr}\ \texttt{in}\ a:\tau\Rightarrow\\ \rho(k':\zeta(\alpha)[\mathcal{L}'])\texttt{open}\ (\rho(k:\zeta(\alpha)[\mathcal{L}])\texttt{pack}\ \lambda(x:\tau')\langle a'\rangle_{k'}\\ \texttt{as}\ \exists(k_0)(\tau'\to\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\setminus\mathcal{K}'\cup\{k_0,k'\}})\ \texttt{hiding}\ \mathcal{K}')\ \texttt{as}\ [k'',c]\ \texttt{in}\ \overline{a}\end{array}}$$

(New)

$$\frac{\begin{array}{c}(c:(\mathcal{K}',\tau'\to\zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}))\in\Gamma\\ \Gamma\vdash\zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}\preceq\zeta(\alpha)[\mathcal{L}]_{\mathcal{K}\setminus\mathcal{K}'}\end{array}}{\Gamma\vdash \texttt{new}\ c:\tau'\to\zeta(\alpha)[\mathcal{L}]_{\mathcal{K}\setminus\mathcal{K}'}\Rightarrow c}$$

Figure 9: Translation of expressions involving classes

and NEW. In the translation of a class binding, a view k is introduced locally. It is used in the translation a' of the definition *expr* of the class. The expression *expr* is therefore translated in the environment Γ in which the class parameter x and the view k have been added. The type of this expression tells us that the set of views $\mathcal{K}'$ (containing in particular k because of rule LOCAL-VIEW) must be hidden from the type $\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}}$ of the definition. This is achieved by packing the definition of the class. Then, the definition is unpacked, introducing in the remainder of the program another view k, abstract this time, replacing $\mathcal{K}'$, and a binding to the class c. This variable is introduced in the environment of the second premise with a type indicating that the view k will have to be hidden from the type of the class when creating an object from this class (or inheriting of it). And indeed, the rule NEW ensures that the view is hidden by subtyping at this time, enforcing the fact that private views are not visible in objects.

3.3 Soundness of the translation

An expression typed in an environment Γ is translated into an expression that can be typed in the environment $\overline{\Gamma}$ defined as the environment Γ where each class binding $c : (\mathcal{K}, \tau)$ is replaced by the binding $c : \tau$. We have proved the soundness of this translation:

Theorem 3 (Translation soundness)
If $\Gamma \vdash a : \tau \Rightarrow \overline{a}$, *then* $\overline{\Gamma} \vdash \overline{a} : \tau$.

The proof is given in appendix A.

3.4 Example

We illustrate the translation on the class `comparable` below. This is the same class as the one given in section 1.2, but with the syntactic sugar expanded.

$\texttt{class comparable}\,[\texttt{protected}\ k](\texttt{x0} : \texttt{int}) = \texttt{object}$
$\quad\texttt{abstract x} : \varsigma(\alpha)\texttt{int}$
$\quad\texttt{abstract equal} : \varsigma(\alpha)\alpha \to \texttt{bool}$
$\quad\texttt{method x} = \varsigma(\texttt{self} : \alpha)\texttt{x0}$
$\quad\texttt{method equal} =$
$\qquad\varsigma(\texttt{self} : \alpha)\lambda(\texttt{other} : \alpha)\texttt{self}._k\texttt{x} = \texttt{other}._k\texttt{x}$
$\texttt{end}$

The class is encoded as a function taking an argument `x0` and returning an object. This object is constructed starting from the empty object. All the methods of the class are first allocated (this correspond to the `abstract` clauses). Then, the view k is captured and the methods are defined. The view is used by the method `equal` to access the other methods of the class. The class is packed, so as to make its view abstract. Finally, the `open` construction makes the class and its view available to the remainder of the program.

$$
\begin{array}{l}
\texttt{open} \\
\quad \rho(k : \zeta(\alpha)[\texttt{x} : \texttt{int}; \texttt{equal} : \alpha \rightarrow \texttt{bool}]) \\
\quad\quad \texttt{pack} \\
\quad\quad\quad \lambda(\texttt{x0} : \texttt{int}) \\
\quad\quad\quad\quad \langle \zeta(\texttt{s} : \alpha)[\emptyset : \emptyset]^{\emptyset}_{\emptyset} \\
\quad\quad\quad\quad\quad + (\texttt{x} : \varsigma(\alpha)\texttt{int}) \\
\quad\quad\quad\quad\quad + (\texttt{equal} : \varsigma(\alpha)\alpha \rightarrow \texttt{bool})\rangle_k \\
\quad\quad\quad\quad\quad .\texttt{x} \Leftarrow \varsigma(\texttt{s} : \alpha)\texttt{x0} \\
\quad\quad\quad\quad\quad .\texttt{equal} \Leftarrow \varsigma(\texttt{s} : \alpha)\lambda(\texttt{other} : \alpha)\texttt{s}._k\texttt{x} = \texttt{other}._k\texttt{x} \\
\quad\quad \texttt{as } \exists(k')\texttt{int} \rightarrow \zeta(\alpha)[\texttt{get\_x} : \texttt{int}; \texttt{equal} : \alpha \rightarrow \texttt{bool}]^{\emptyset}_{\{k'\}} \\
\quad\quad \texttt{hiding } \{k\} \\
\texttt{as } [k, \texttt{comparable}] \texttt{ in} \ldots
\end{array}
$$

4 Friend functions

A friend function is a function that has a privileged access to objects of a class. The existence of such privileged access is of particular importance for privacy. Indeed, if for instance two objects needed to interact between each other, all methods necessary for this interaction would otherwise have to remain public. The usual trick to encode friend functions in a structural typing setting is to use a method `repr` that returns the whole internal state of the object [PT93]. So that only friend functions can access this state, the type of this method is hidden using an abstract type. However, this technique defeats our goal of being able to hide arbitrary methods. Indeed, if the method is hidden in a subclass, the friend function will not be able to manipulate objects of this subclass. Another possibility is to embed the class and its friend functions into a module than does not export the class, but only a constructor for the objects of the class [FR99]. Any method of the class can be hidden in the type of the constructor by making it partially abstract. However, this technique precludes any further subclassing. Thus, both these solutions are unsatisfactory. We show how public views can be used to implement friend functions without any of these limitations.

Let us consider the following example.

```
module M : sig
  class c [protected k] : object end
  val f : c -> int
end = struct
  class c [public k] (x0 : int) = object
    method x : int = x0
  end
  let f (v : Obj('a)[ | k ]) = obj.[k]x
end
```

The class `c` introduces a public view. This view is used by the function `f` in order to access the method `x` of objects holding this view. The view is then made abstract and the method `x` is hidden by signature constraint. However, it is still possible to apply the function `f` to an object of class `c`, even though no method is exported. Moreover, the class `c` can be freely extended by inheritance, and the function `f` can be applied to objects of its subclasses, as long as they inherit the abstract view `k`.

We have not formalized the module system, but we give a possible translation of this example into the core calculus[1].

$$
\begin{array}{l}
\mathtt{open} \\
\quad \rho(k : \zeta(\alpha)[\mathtt{x} : \mathtt{int}]) \\
\quad\quad \mathtt{pack} \\
\quad\quad\quad \mathtt{let\ c} = \\
\quad\quad\quad\quad \lambda(\mathtt{x}_0 : \mathtt{int})\langle \zeta(\mathtt{s} : \alpha)[\emptyset : \emptyset]^{\emptyset}_{\emptyset} + (\mathtt{x} : \varsigma(\alpha)\mathtt{int})\rangle_k \\
\quad\quad\quad\quad\quad\quad\quad\quad .\mathtt{x} \Leftarrow \varsigma(\mathtt{s} : \alpha)\mathtt{x}_0\ \mathtt{in} \\
\quad\quad\quad \mathtt{let}\ f = \lambda(\mathtt{v} : \zeta(\alpha)[\emptyset]_{\{k\}})\mathtt{v}._k\mathtt{x}\ \mathtt{in} \\
\quad\quad\quad \zeta(\mathtt{s} : \beta)[\emptyset : \emptyset]^{\emptyset}_{\emptyset} \\
\quad\quad\quad\quad + (\mathtt{c} : \varsigma(\beta)\zeta(\alpha)[\emptyset]^{\emptyset}_{\{k\}}) \\
\quad\quad\quad\quad .\mathtt{c} \Leftarrow \varsigma(\mathtt{s} : \beta)\lambda(\mathtt{x}_1 : \mathtt{int})(\mathtt{c}(\mathtt{x}_1) \setminus \mathtt{x}) \\
\quad\quad\quad\quad + (\mathtt{f} : \varsigma(\beta)\zeta(\alpha)[\emptyset]_{\{k\}} \to \mathtt{int}) \\
\quad\quad\quad\quad .\mathtt{f} \Leftarrow \varsigma(\mathtt{s} : \beta)\mathtt{f} \\
\quad\quad \mathtt{as}\ \exists(k')\zeta(\beta)[\mathtt{c} : \zeta(\alpha)[\emptyset]^{\emptyset}_{\{k'\}}; \mathtt{f} : \zeta(\alpha)[\emptyset]_{\{k'\}} \to \mathtt{int}]^{\emptyset}_{\emptyset} \\
\quad\quad \mathtt{hiding}\ \{k\} \\
\mathtt{as}\ [k, \mathtt{m}]\ \mathtt{in} \ldots
\end{array}
$$

The view k is the public view of class `c`. It is introduced just before the definition of the structure, so as to be visible in the whole definition. Then, the class and the function are defined (variables `c` and `f`). The module is represented as an object. The methods of this object are filled with the components of the structure, coerced to their final types. For that, the class `c` is wrapped in a coercion function which hides the method `x`.

5 Typing issues

The calculus has been designed so as to make type checking straightforward. A consequence of this is that the subtyping rule SUB-OBJECT is more restrictive than what type soundness alone would have required. Indeed, this rule does not allow the hiding of methods when selftype is not covariant in a view of the object. If this restriction was removed, the calculus would not have principal types. Indeed, let us consider for instance an object x of type

$$\tau_0 = \zeta(\alpha)[n : \mathtt{int}]_{\{k\}}$$

where the view k, has type $\zeta(\alpha)[m : \alpha \to \alpha]$. Then $x._k m$ has type $\tau_0 \to \tau_0$. If subtyping were allowed in this case, x would also have type

$$\tau_1 = \zeta(\alpha)[\ ]_{\{k\}}.$$

Then, $x._k m$ would also have type $\tau_1 \to \tau_1$, which is not comparable with type $\tau_0 \to \tau_0$. This difficulty could be avoided using higher-order types: indeed, the type of $x._k m$ could be $\alpha \to \alpha$ for all α such that $\tau_0 \preceq \alpha \preceq \tau_1$. Hiding arbitrary

[1] In this translation, we use a construction `let` which we have not formalized. It can be considered simply as syntactic sugar: `let` $x = a'$ `in` a is the expression a in which all occurences of x have been replaced by a'.

method in a proper object with a binary method would also be safe, but raises the same difficulty. This difficulty is actually the reason why we had to make a distinction between prototypes and proper objects in the calculus: we want to be able to hide arbitrary method in prototypes.

We believe that the typing rules could be easily adapted in order to make type inference possible, using row variables [Wan87] in a similar way to Objective Caml [RV98, Rém94]. Of course, subtyping would have to be explicit, but according to our experience with Objective Caml, this is not a problem in practice. Furthermore, this would allow us to get rid of the restrictions expose in the paragraph above. The type of an object would be composed of two possibly extensible rows: one for methods, the other one for views. We expect that the typing of self would then not require any special rule. Indeed, selftype could be modeled as an object type with no method and an extensible row of views.

For the sake of simplicity, we have only considered width subtyping for objects in our calculus. However, we expect that the addition of depth subtyping should be straightforward and only require minor adjustments to the soundness proof. An unfortunate consequence of the limitation to width subtyping is that variables occuring in an object type are non-variant. With depth-subtyping, the definition of covariance could be updated so as to get rid of this restriction.

6 Anomaly with multiple inheritance

While this calculus can be used for a class-based language with single inheritance, it unfortunately exhibits an anomaly that makes problematic its use as a base for a language with multiple inheritance. Indeed, abstraction can be broken under certain circumstance.

Indeed, let us consider an object with a view k. This view can be hidden from the type of the object using an abstract view. One could think that there is then no way to use this view of the object from outside anymore. In particular, one would expect that the methods that can only be access via this view cannot be redefined, and that the dictionary associated to this view in the object cannot be changed. However, if the view is still available in the environment, nothing prevent us from adding it again to the object. This will change the dictionary associated to the view in the object: it will become the same as the dictionary of the primary interface of the object. And methods can then be redefined using this view.

In a language with single inheritance, the scope of the views can be controlled so as to avoid this problem. With multiple inheritance, however, if a class inherits twice from another class (either directly or indirectly), it will inherit from its views twice. So, if a view have been hidden on only one of the inheritance paths, it will still be visible in the class.

7 Related work

Riecke and Stone [RS00] have designed an object calculus with object extension where methods can be arbitrarily hidden. This is achieved using subsumption and by a privileged access to other methods of self from a method definition. Their calculus does not handle binary methods. Our calculus is strongly inspired by this work. Indeed, we reuse the idea of dictionaries for decoupling the internal naming from the external naming of methods. Apart from its strong connection, the calculi use completely different mechanisms to achieve their goal.

Moby [FR98] is an experimental object-oriented language. It aims at providing a good interaction between classes and modules. Like our language, it allows arbitrary method hiding in classes. It is based on a variant of the first-order version of Riecke and Stone calculus with stateful objects. As a consequence, it does not support binary methods. Moreover, being first order, it does not even have the notion of selftype. An extension of Moby with inheritance-based subtyping has been recently proposed by Fisher and Reppy [FR00]. The calculus they have developed to validate their design (XMOC) is class-based. Classes play in this calculus a role similar to views in our calculus. They propose in this paper a solution for friend functions (written in Extended Moby) which is very close to the one of section 4. However, their solution is not formalized, and it appears that it cannot be faithfully encoded in XMOC. Indeed, this calculus does not provide any abstraction mechanism beyond privacy by subsumption in classes. Therefore, it cannot model the fact that a function has a privileged access to some methods of an object (that is, only it can access these methods).

Several calculi combining object extension with object subsumption have been proposed, starting with the work of Fisher and Mitchell [FM95]. They ensure soundness by having two kind of object types, prototypes and proper objects, as our calculus. They usually do not allow method hiding in prototypes. This is the case for the calculus presented by Gianantonio, Honsel and Liquori in [GHL98], as well as the one proposed by Bono, Bugliesi, Liquori and Dezani-Ciancaglini in [BBLDC97]. In the calculus presented by Bono and Fisher in [BF98], it is possible to hide the type of any methods of a prototype. This is achieved by wrapping prototypes within existentials. Our calculus use the same idea to hide views (this idea is initially due to Pierce and Turner [PT93]). However, the methods are not completely hidden: their names remain visible and the prototype cannot be extended with new methods of the same name.

A need for context-dependent behavior of objects has emerged in the context of database languages: depending on the context, an object would play different roles, while retaining its identity. This notion of *roles* has been formalized by Ghelli and Palmerini [GP99]. They present a first-order calculus with roles. It appears that views are similar to roles though they have been introduced for a completely different purpose. Contrary to our calculus, however, each method of an object belongs to exactly one role. Another difference is that the roles form a hierarchy, and at the invocation of a method l, the method selected is the method associated to l in the minimum super-role of the current role.

Flatt, Krishnamurthi and Felleisen [FKF98] make use of views to define powerful mixins (a mixin is a function from classes to classes). But their language seems to rely heavily on cast to achieve this expressiveness, at the expense of type safety.

8 Conclusion

This paper presents an object-oriented calculus allowing arbitrary hiding of methods in prototypes. This calculus also has binary methods and admits a faithful encoding of friend functions. We believe that this is the first calculus smoothly combining all these features. This demonstrates that good modularity properties of an object-oriented language does not necessarily come at the expense of its expressiveness. Moreover, we designed a class-based surface language as a minor extension, thus showing the practicality of the calculus. There actually exists a simple typed translation of the extension into the core calculus.

Views are a key ingredient for the soundness of our calculus. They are also an interesting feature for an object-oriented language. Indeed, they bring more flexibility by allowing an object to simultaneously possess several distinct interfaces. We therefore believe that this notion could be used fruitfully for other purposes.

Our calculus does not currently handle multiple inheritance. We hope that multiple inheritance is possible even though this might requires a significantly different semantics. This is another important direction for future work.

References

[BBLDC97] V. Bono, M. Bugliesi, L. Liquori, and M. Dezani-Ciancaglini. Subtyping constraint for incomplete objects. In *Proceedings of TAPSOFT/CAAP*, number 1214 in LNCS, pages 465–477. Springer-Verlag, 1997.

[BCC+95] Kim B. Bruce, Luca Cardelli, Giuseppe Castagna, The Hopkins Object Group, Gary T. Leavens, and Benjamin Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1995.

[BF98] Viviana Bono and Kathleen Fisher. An imperative, first-order calculus with object extension. In *ECOOP*, pages 462–497, 1998.

[BFP97] Kim B. Bruce, Adrian Fiech, and Leaf Petersen. Subtyping is not a good "match" for object-oriented languages. In *ECOOP*, number 1241 in LNCS, pages 104–127. Springer-Verlag, 1997.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction and polymorphism. *ACMCS*, 17(4):471–522, December 1985.

[FKF98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Conference Record of POPL '98: The 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183, San Diego, California, 19–21 January 1998.

[FM95] K. Fisher and J.C. Mitchell. A delegation-based object calculus with subtyping. In *Fundamentals of Computation Theory (FCT'95)*, number 965 in LNCS, pages 42–61. Springer-Verlag, 1995.

[FR98] Kathleen Fisher and John Reppy. Foundations for Moby classes. Bell Labs Technical Memorandum, December 1998.

[FR99] Kathleen Fisher and John Reppy. The design of a class mechanism for Moby. In ACM Press, editor, *Conference on Programming Language Design and Implementation*, pages 37–49, 1999.

[FR00] Kathleen Fisher and John Reppy. Extending Moby with inheritance-based subtyping. In *ECOOP*, 2000. To appear.

[GHL98] Pietro Di Gianantonio, Furio Honsell, and Luigi Liquori. A lambda calculus of objects with self-inflicted extension. In ACM SIGPLAN Notices, editor, *Proceedings of the conference on Object-oriented programming, systems, languages, and applications*, pages 166–178, October 1998.

[GP99] Giorgio Ghelli and Debora Palmerini. Foundations for extensible objects with roles. Presented at the FOOL'6 workshop, January 1999.

[LVD+96] Xavier Leroy, Jérôme Vouillon, Damien Doligez, et al. The Objective Caml system. Software and documentation available on the Web, `http://pauillac.inria.fr/ocaml/`, 1996.

[MPW92] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Information and Computation*, 100(1):1–40, September 1992.

[PT93] Benjamin C. Pierce and David N. Turner. Statically typed friendly functions via partially abstract types. Technical Report ECS-LFCS-93-256, University of Edinburgh, LFCS, April 1993. Also available as INRIA-Rocquencourt Rapport de Recherche No. 1899.

[Rém94] 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, 1994.

[RS00] Jon G. Riecke and Christopher A. Stone. Privacy via subsumption. *Information and Computation*, 2000. To appear.

[RV98] Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Object Systems*, 4(1):27–50, 1998. A preliminary version appeared in the proceedings of the 24th ACM Conference on Principles of Programming Languages, 1997.

[Wan87] Mitchell Wand. Complete type inference for simple objects. In D. Gries, editor, *Second Symposium on Logic In Computer Science*, pages 207–276, Ithaca, New York, June 1987. IEEE Computer Society Press.

[WF91] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. Technical Report COMP TR91-160, Department of Computer Science, Rice University, Houston, Texas, April 1991.

A Proofs

A.1 Subject reduction

We start by proving a few lemma used in the proof.

Lemma 4 (Decomposition and replacement) *If $\Gamma \vdash E[a] : \tau$, then $\Gamma' \vdash a : \tau'$ for some environment Γ' and some type τ'. Furthermore, if $\Gamma' \vdash a' : \tau'$ then $\Gamma \vdash E[a'] : \tau$.*

Proof. By induction on the proof of $\Gamma \vdash E[a] : \tau$. ∎

Lemma 5 (Context strengthening) *If $\Gamma \vdash \mathcal{J}$ and $\Gamma \subseteq \Gamma'$, then $\Gamma' \vdash \mathcal{J}$ (where $\mathcal{J}$ stand for any right-hand side of a judgement).*

Proof. By induction on the proof of $\Gamma \vdash \mathcal{J}$ ∎

Lemma 6 (Matching strengthening) *If $\Gamma; (\alpha <\# \mathcal{K}) \vdash a : \tau$ and $\mathcal{K} \subseteq \mathcal{K}'$ then $\Gamma; (\alpha \preceq \mathcal{K}') \vdash a : \tau$.*

Proof. By induction on the proof of the first premise.

An assumption $\alpha <\# \mathcal{K}$ may only be used by rule Match-Var:

$$\frac{(\alpha <\# \mathcal{K}) \in \Gamma \qquad \mathcal{K}_1 \subseteq \mathcal{K}}{\Gamma \vdash \alpha <\# \mathcal{K}_1} \ (\textsc{Match-Var})$$

By transitivity, $\mathcal{K}_1 \subseteq \mathcal{K}'$. Therefore:

$$\frac{(\alpha <\# \mathcal{K}) \in \Gamma \qquad \mathcal{K}_1 \subseteq \mathcal{K}'}{\Gamma \vdash \alpha <\# \tau_1} \ (\textsc{Match-Var})$$

Other cases are easy. ∎

Lemma 7 (Matching elimination) *If $\Gamma; (\alpha <\# \mathcal{K}) \vdash a : \tau$ and $\Gamma' \vdash \tau' <\# \mathcal{K}$ where $\Gamma' \subseteq \Gamma$ then $\Gamma\{\tau'/\alpha\} \vdash a\{\tau'/\alpha\} : \tau\{\tau'/\alpha\}$.*

Proof. By induction on the proof of the first premise. We simultaneously prove that if $\Gamma; (\alpha <\# \mathcal{K}) \vdash \tau_1 \preceq \tau_2$ and $\Gamma' \vdash \tau' \preceq \mathcal{K}$ where $\Gamma' \subseteq \Gamma$ then $\Gamma\{\tau'/\alpha\} \vdash \tau_1\{\tau'/\alpha\} \preceq \tau_2\{\tau'/\alpha\}$.

The most difficult case is the case of rule Match-Var.

$$\frac{(\alpha_1 <\# \mathcal{K}_1) \in \Gamma; (\alpha <\# \mathcal{K}) \qquad \mathcal{K}' \subseteq \mathcal{K}_1}{\Gamma; (\alpha <\# \mathcal{K}) \vdash \alpha_1 <\# \mathcal{K}'} \ (\textsc{Match-Var})$$

If $\alpha_1 \neq \alpha$, we have $(\alpha_1 <\# \mathcal{K}_1) \in \Gamma\{\tau'/\alpha\}$. Hence, $\Gamma\{\tau'/\alpha\} \vdash \alpha_1\{\tau'/\alpha\} <\# \mathcal{K}'$ as desired.

Let us now assume that $\alpha_1 = \alpha$. Then, $\mathcal{K}_1 = \mathcal{K}$. As α does not appear free in the second premise, $\Gamma'\{\tau'/\alpha\} \vdash \tau' <\# \mathcal{K}$. By lemma 5, $\Gamma\{\tau'/\alpha\} \vdash \tau' <\# \mathcal{K}$. By inspection of the matching rules, using the fact that $\mathcal{K}' subseteq \mathcal{K}$, we conclude that $\Gamma\{\tau'/\alpha\} \vdash \tau' <\# \mathcal{K}'$ as desired.

Other cases are easy. ∎

Lemma 8 (Abstract view elimination) *If $\Gamma; (k) \vdash a : \tau$ and $\mathcal{K} \subseteq \operatorname{dom}\Gamma$ then $\Gamma\{\mathcal{K}/k\} \vdash a\{\mathcal{K}/k\} : \tau\{\mathcal{K}/k\}$.*

Proof. By induction on the proof of the first premise.

- Rule PROTO:

$$\frac{\begin{array}{ll} \mathcal{L}' = \mathcal{L} \circ \varphi & \mathcal{A} = \operatorname{dom} \mathcal{L}' \setminus \operatorname{dom}(L \circ \varphi) \\ \mathcal{K}' = \operatorname{dom} \Phi & \operatorname{dom} \mathcal{L} \subseteq \operatorname{dom} L \cup \operatorname{range} \varphi \\ \multicolumn{2}{l}{\text{For all } l \text{ in } \operatorname{dom} L,\ \Gamma; (\alpha <\# \mathcal{K}); (x : \alpha) \vdash L(l) : \mathcal{L}(l)} \\ \multicolumn{2}{l}{\forall k' \in \operatorname{dom} \Phi \cdot (k' : \zeta(\alpha)[\mathcal{L}'']) \in \Gamma; (k) \wedge \mathcal{L}'' = \mathcal{L} \circ \Phi(k)} \end{array}}{\Gamma; (k) \vdash \zeta(x : \alpha)[L : \mathcal{L}]^{\varphi}_{\Phi} : \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}'}}$$

  For all $k' \in \mathcal{K}$, $(k' : \zeta(\alpha)[\mathcal{L}'']) \in \Gamma; (k)$. As a consequence, $\mathcal{K}'\{\mathcal{K}'/k\} = \mathcal{K}'$. By induction hypothesis, for all l in $\operatorname{dom} L$, $\Gamma\{\mathcal{K}/k\}; (\alpha <\# \mathcal{K}'); (x : \alpha) \vdash L(l) : \mathcal{L}(l)$. All other premises of this rule are clearly preserved by the substitution. So, an application of the rule VIEW-SELECTION allows us to conclude this case.

- $$\frac{\begin{array}{ll} \Gamma; (k) \vdash a : \tau & \Gamma; (k) \vdash \tau <\# \mathcal{K}' \\ k' \in \mathcal{K}' & (k' : \zeta(\alpha)[\mathcal{L}']) \in \Gamma; (k) \end{array}}{\Gamma; (k) \vdash a ._{k'} l : (\mathcal{L}'(l))\{\tau/\alpha\}} \ (\text{VIEW-SELECTION})$$

  We have $(k' : \zeta(\alpha)[\mathcal{L}']) \in \Gamma; (k)$. Therefore $k' \neq k$. Then, we have $k' \in \mathcal{K}'\{\mathcal{K}/k\}$. By induction hypothesis, $\Gamma\{\mathcal{K}/k\} \vdash \tau\{\mathcal{K}/k\} \preceq \zeta(\alpha)[\emptyset]_{\mathcal{K}'}\{\mathcal{K}/k\}$. All other premises of this rule are clearly preserved by the substitution. So, an application of the rule VIEW-SELECTION allows us to conclude this case.

- Rules VIEW-OVERRIDE, VIEW-CAPTURE and VIEW-REPLACE: the proof is similar to the previous case.

- $$\frac{\Gamma; (k) \vdash a : \tau\{\mathcal{K}'/k'\}}{\Gamma; (k) \vdash \texttt{pack}\ a\ \texttt{as}\ \exists(k')\tau\ \texttt{hiding}\ \mathcal{K}' : \exists(k')\tau} \ (\text{PACK})$$

  By renaming, we can assume that $k' \neq k$ and $k' \notin \mathcal{K}$. By induction hypothesis, $\Gamma\{\mathcal{K}/k\} \vdash a\{\mathcal{K}/k\} : \tau\{\mathcal{K}'/k'\}\{\mathcal{K}/k\}$. From the preceding remark, $\tau\{\mathcal{K}'/k'\}\{\mathcal{K}/k\} = \tau\{\mathcal{K}/k\}\{\mathcal{K}'\{\mathcal{K}/k\}/k'\}$. As a consequence, using rule PACK, $\Gamma\{\mathcal{K}/k\} \vdash \texttt{pack}\ a\{\mathcal{K}/k\}\ \texttt{as}\ \exists(k')(\tau\{\mathcal{K}/k\})\ \texttt{hiding}\ \mathcal{K}'\{\mathcal{K}/k\} : \exists(k')\tau\{\mathcal{K}/k\}$. That is, $\Gamma\{\mathcal{K}/k\} \vdash (\texttt{pack}\ a\ \texttt{as}\ \exists(k')\tau\ \texttt{hiding}\ \mathcal{K}')\{\mathcal{K}/k\} : (\exists(k')\tau)\{\mathcal{K}/k\}$ as desired.

- $$\frac{\begin{array}{c} \mathcal{L}' \subseteq \mathcal{L} \qquad \mathcal{K}'' \subseteq \mathcal{K}' \\ \alpha \text{ appears covariantly in } \mathcal{L}' \end{array}}{\Gamma; (k) \vdash \zeta(\alpha)[\mathcal{L}]_{\mathcal{K}'} \preceq \zeta(\alpha)[\mathcal{L}']_{\mathcal{K}''}} \ (\text{SUB-OBJECT})$$

  We have $\mathcal{K}''\{\mathcal{K}/k\} \subseteq \mathcal{K}'\{\mathcal{K}/k\}$. Other premises are clearly preserved by the substitution. So, an application of the rule VIEW-SELECTION allows us to conclude this case.

- Other cases are easy.

■

Lemma 9 (Value substitution) *If* $\Gamma; (x : \tau) \vdash a' : \tau'$ *and* $\Gamma \vdash a : \tau$*, then* $\Gamma \vdash a'\{a/x\} : \tau'$*.*

Proof. By induction on the proof of the first premise.
This proof is standard; there is no specific difficulty. ∎

Lemma 10 (Subtyping transitivity) *If* $\Gamma \vdash \tau \preceq \tau'$ *and* $\Gamma \vdash \tau' \preceq \tau''$ *then* $\Gamma \vdash \tau \preceq \tau''$.

Proof. By cases on the final rules of derivations proving the premises.

- Rule Sub-Refl and any: Trivial.
- Rule Sub-Match then Sub-Object: We have $\tau = \alpha$ and $\tau' = \zeta(\alpha')[\emptyset]_{\mathcal{K}}$ and $\tau'' = \zeta(\alpha')[\emptyset]_{\mathcal{K}'}$ with $\mathcal{K}' \subseteq \mathcal{K}$. By inspection of the matching rules, we see that one can directly deduce $\tau \preceq \tau''$.
- Rules Sub-Arrow and Sub-Arrow: By direct application of the induction hypothesis.
- Rules Sub-Object and Sub-Object: By transitivity of set inclusion.

∎

Lemma 11 (Class subtyping) *If* $\Gamma \vdash \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}} \preceq \zeta(\alpha)[\mathcal{L}']_{\mathcal{K}'}$, *then* $\mathcal{A} = \emptyset$, $\mathcal{L}' \subseteq \mathcal{L}$ *and* $\mathcal{K}' \subseteq \mathcal{K}$.

Proof. By induction on the proof of the premise. ∎

Lemma 12 (Covariant substitution) *If* α *appears covariantly in* τ *and* $\Gamma \vdash \tau' \preceq \tau''$, *then* $\Gamma \vdash \tau\{\tau'/\alpha\} \preceq \tau\{\tau''/\alpha\}$.

Proof. By induction on the proof of the second premise. We prove simultaneously the statement above and the following statement:

> *If* α *appears contravariantly in* τ *and* $\Gamma \vdash \tau' \preceq \tau''$, *then* $\Gamma \vdash \tau\{\tau''/\alpha\} \preceq \tau\{\tau'/\alpha\}$.

Let us first consider the first statement. If α does not appear free in τ, then the conclusion follows from Sub-Refl. If $\tau = \alpha$, then the conclusion is given by the second assumption $\Gamma \vdash \tau' \preceq \tau''$. Otherwise, $\tau = \tau_1 \to \tau_2$ where α appears contravariantly in τ_1 and covariantly in τ_2. By induction hypothesis, $\Gamma \vdash \tau_1\{\tau''/\alpha\} \preceq \tau_1\{\tau'/\alpha\}$ and $\Gamma \vdash \tau_2\{\tau'/\alpha\} \preceq \tau_2\{\tau''/\alpha\}$. We conclude by rule Sub-Arrow.

Let us now consider the second statement. If α does not appear free in τ, then the conclusion follows from Sub-Refl. Otherwise, $\tau = \tau_1 \to \tau_2$ where α appears covariantly in τ_1 and contravariantly in τ_2. By induction hypothesis, $\Gamma \vdash \tau_1\{\tau'/\alpha\} \preceq \tau_1\{\tau''/\alpha\}$ and $\Gamma \vdash \tau_2\{\tau''/\alpha\} \preceq \tau_2\{\tau'/\alpha\}$. We conclude by rule Sub-Arrow. ∎

We can now start the subject reduction proof.

Proof. By cases depending on the operational rule used.

Because of the transitivity (lemma 10) and reflexivity of subtyping, we may assume without loss of generality that in the proof of $\Gamma \vdash a : \tau$, uses of the subsumption rule Sub alternate with the use of other typing rules. It is also sufficient to only consider the cases where this proof does not end with a use of subsumption. Lastly, by lemma 4, we need only consider the local reduction steps:

- $\zeta(x:\alpha)[L:\mathcal{L}]^{\varphi}_{\Phi} + (l : \varsigma(\alpha)\tau) \longrightarrow \zeta(x:\alpha)[L:(\mathcal{L};(l':\tau))]^{\varphi;(l=l')}_{\Phi}$ where $l' \notin \operatorname{dom}\mathcal{L} \cup \operatorname{range}\varphi$.

  The proof of the first assumption must end in a use of rule Extension. Thus, we have $\Gamma \vdash \zeta(x:\alpha)[L:\mathcal{L}]^{\varphi}_{\Phi} : \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}}$ for some $\mathcal{L}'$, $\mathcal{A}$ and $\mathcal{K}$ such that $\tau = \zeta(\alpha)[\mathcal{L}';(l:\tau)]^{\mathcal{A}\cup\{l\}}_{\mathcal{K}}$. This judgment can be proved using the rule Proto. Hence,

$$\begin{cases} \mathcal{L}' = \mathcal{L} \circ \varphi \\ \mathcal{A} = \operatorname{dom}\mathcal{L}' \setminus \operatorname{dom}(L \circ \varphi) \\ \mathcal{K} = \operatorname{dom}\Phi \\ \operatorname{dom}\mathcal{L} \subseteq \operatorname{dom} L \cup \operatorname{range}\varphi \\ \text{For all } l \text{ in } \operatorname{dom} L,\ \Gamma; (\alpha <\# \mathcal{K}); (x:\alpha) \vdash L(l) : \mathcal{L}(l) \\ \forall k \in \operatorname{dom}\Phi \cdot (k : \zeta(\alpha)[\mathcal{L}'']) \in \Gamma \wedge \mathcal{L}'' = \mathcal{L} \circ \Phi(k) \end{cases}$$

  Let $\varphi_0 = (\varphi;(l=l'))$, $\mathcal{L}_0 = (\mathcal{L};(l'=\tau))$, $\mathcal{L}'_0 = (\mathcal{L}';(l=\tau))$ and $\mathcal{A}_0 = \mathcal{A}\cup\{l\}$. We are going to prove that the hypotheses above still hold after a substitution of φ_0 for φ, $\mathcal{L}_0$ for $\mathcal{L}$, $\mathcal{L}'_0$ for $\mathcal{L}'$ and $\mathcal{A}_0$ for $\mathcal{A}$. This will complete the proof of this case, as it entails $\Gamma \vdash \zeta(x:\alpha)[L:\mathcal{L}_0]^{\varphi_0}_{\Phi} : \zeta(\alpha)[\mathcal{L}'_0]^{\mathcal{A}_0}_{\mathcal{K}}$, that is, $\Gamma \vdash a' : \tau$.

$$\begin{array}{rcll} \mathcal{L}_0 \circ \varphi_0 & = & (\mathcal{L};(l'=\tau)) \circ (\varphi;(l=l')) & \\ & = & ((\mathcal{L};(l'=\tau)) \circ \varphi);(l=\tau) & \\ & = & (\mathcal{L} \circ \varphi);(l=\tau) & (l' \notin \operatorname{dom}\mathcal{L} \cup \operatorname{range}\varphi) \\ & = & \mathcal{L}';(l=\tau) & \\ & = & \mathcal{L}'_0 & \end{array}$$

$$\begin{array}{rcl} \operatorname{dom}\mathcal{L}'_0 \setminus \operatorname{dom}(L \circ \varphi_0) & = & \operatorname{dom}(\mathcal{L}';(l=\tau)) \setminus \operatorname{dom}(L \circ (\varphi;(l=l'))) \\ & = & (\operatorname{dom}\mathcal{L}' \cup \{l\}) \setminus \operatorname{dom}(L \circ (\varphi;(l=l'))) \\ & = & (\operatorname{dom}\mathcal{L}' \cup \{l\}) \setminus (\operatorname{dom}(L \circ \varphi) \setminus \{l\}) \\ & & \qquad (l' \notin \operatorname{dom}\mathcal{L} \text{ and } \operatorname{dom} L \subseteq \operatorname{dom}\mathcal{L}) \\ & = & (\operatorname{dom}\mathcal{L}' \setminus \operatorname{dom}(L \circ \varphi)) \cup \{l\} \\ & = & \mathcal{A} \cup \{l\} \\ & = & \mathcal{A}_0 \end{array}$$

$$\begin{array}{rcl} \operatorname{dom} L \cup \operatorname{range}\varphi_0 & = & \operatorname{dom} L \cup \operatorname{range}(\varphi;(l=l')) \\ & = & \operatorname{dom} L \cup \operatorname{range}\varphi \cup \{l'\} \\ & \supseteq & \operatorname{dom}\mathcal{L} \cup \{l'\} \\ & \supseteq & \operatorname{dom}\mathcal{L}_0 \end{array}$$

  Finally, as $l' \notin \operatorname{dom}\mathcal{L}$, $\mathcal{L} \subseteq \mathcal{L}_0$. Therefore, the two last premises still hold after the substitution.

- $\zeta(x:\alpha)[L:\mathcal{L}]^{\varphi}_{\Phi} \setminus l \longrightarrow \zeta(x:\alpha)[L:\mathcal{L}]^{\varphi|_{\operatorname{dom}\varphi\setminus\{l\}}}_{\Phi}$

  The proof of the first assumption must end in a use of rule Restriction. Thus, we have $\Gamma \vdash \zeta(x:\alpha)[L:\mathcal{L}]^{\varphi}_{\Phi} : \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}}$ for some $\mathcal{L}'$, $\mathcal{A}$ and $\mathcal{K}$

such that $\tau = \zeta(\alpha)[\mathcal{L}'|_{\mathrm{dom}\,\mathcal{L}'\setminus\{l\}}]^{\mathcal{A}}_{\mathcal{K}}$ and $l \in \mathrm{dom}\,\mathcal{L}' \setminus \mathcal{A}$. This can be proved using the rule PROTO. Hence,

$$\begin{cases} \mathcal{L}' = \mathcal{L} \circ \varphi \\ \mathcal{A} = \mathrm{dom}\,\mathcal{L}' \setminus \mathrm{dom}(L \circ \varphi) \\ \mathcal{K} = \mathrm{dom}\,\Phi \\ \mathrm{dom}\,\mathcal{L} \subseteq \mathrm{dom}\,L \cup \mathrm{range}\,\varphi \\ \text{For all } l \text{ in } \mathrm{dom}\,L,\ \Gamma; (\alpha <\# \mathcal{K}); (x : \alpha) \vdash L(l) : \mathcal{L}(l) \\ \forall k \in \mathrm{dom}\,\Phi \cdot (k : \zeta(\alpha)[\mathcal{L}'']) \in \Gamma \wedge \mathcal{L}'' = \mathcal{L} \circ \Phi(k) \end{cases}$$

Let $\varphi_0 = \varphi|_{\mathrm{dom}\,\varphi\setminus\{l\}}$ and $\mathcal{L}'_0 = \mathcal{L}'|_{\mathrm{dom}\,\mathcal{L}'\setminus\{l\}}$. We are going to prove that the hypotheses above still hold after a substitution of φ_0 for φ and $\mathcal{L}'_0$ for $\mathcal{L}$. This will complete the proof of this case, as it entails $\Gamma \vdash \zeta(x : \alpha)[L : \mathcal{L}]^{\varphi_0}_{\Phi} : \zeta(\alpha)[\mathcal{L}'_0]^{\mathcal{A}}_{\mathcal{K}}$, that is, $\Gamma \vdash a' : \tau$.

$$\begin{array}{rcl} \mathcal{L} \circ \varphi_0 & = & \mathcal{L} \circ (\varphi|_{\mathrm{dom}\,\varphi\setminus\{l\}}) \\ & = & (\mathcal{L} \circ \varphi)|_{\mathrm{dom}(\mathcal{L}\circ\varphi)\setminus\{l\}} \\ & = & \mathcal{L}'|_{\mathrm{dom}\,\mathcal{L}'\setminus\{l\}} \\ & = & \mathcal{L}'_0 \end{array}$$

$$\begin{array}{rcll} \mathrm{dom}\,\mathcal{L}'_0 \setminus \mathrm{dom}(L \circ \varphi_0) & = & \mathrm{dom}(\mathcal{L}'|_{\mathrm{dom}\,\mathcal{L}'\setminus\{l\}}) \setminus \mathrm{dom}(L \circ (\varphi|_{\mathrm{dom}\,\varphi\setminus\{l\}})) & \\ & = & (\mathrm{dom}\,\mathcal{L}' \setminus \{l\}) \setminus \mathrm{dom}((L \circ \varphi)|_{\mathrm{dom}(L\circ\varphi)\setminus\{l\}}) & \\ & = & (\mathrm{dom}\,\mathcal{L}' \setminus \{l\}) \setminus (\mathrm{dom}(L \circ \varphi) \setminus \{l\}) & \\ & = & (\mathrm{dom}\,\mathcal{L}' \setminus \mathrm{dom}(L \circ \varphi)) \setminus \{l\} & \\ & = & \mathcal{A} \setminus \{l\} & \\ & = & \mathcal{A} & (l \in \mathrm{dom}\,\mathcal{L}' \setminus \mathcal{A}) \end{array}$$

For the next premise, we will first prove that $\varphi(l) \in \mathrm{dom}\,L$. For that, we have to prove that $l \in \mathrm{dom}(L \circ \varphi)$. We know that $l \in \mathrm{dom}\,\mathcal{L}' \setminus \mathcal{A}$. On the other hand:

$$\begin{array}{rcl} \mathrm{dom}\,\mathcal{L}' \setminus \mathcal{A} & = & \mathrm{dom}\,\mathcal{L}' \setminus (\mathrm{dom}\,\mathcal{L}' \setminus \mathrm{dom}(L \circ \varphi)) \\ & = & \mathrm{dom}\,\mathcal{L}' \cap \mathrm{dom}(L \circ \varphi) \\ & \subseteq & \mathrm{dom}(L \circ \varphi) \end{array}$$

Then:

$$\begin{array}{rcl} \mathrm{dom}\,L \cup \mathrm{range}\,\varphi_0 & = & \mathrm{dom}\,L \cup \mathrm{range}(\varphi|_{\mathrm{dom}\,\varphi\setminus\{l\}}) \\ & = & (\mathrm{dom}\,L \cup \{\varphi(l)\}) \cup \mathrm{range}(\varphi|_{\mathrm{dom}\,\varphi\setminus\{l\}}) \\ & & \qquad (l \in \mathrm{dom}(L \circ \varphi)) \\ & = & \mathrm{dom}\,L \cup \mathrm{range}\,\varphi \\ & \supseteq & \mathrm{dom}\,\mathcal{L} \end{array}$$

Other premises are unchanged.

- $\zeta(x : \alpha)[L : \mathcal{L}]^{\varphi}_{\Phi}.l \Leftarrow \varsigma(x : \alpha)a'' \longrightarrow \zeta(x : \alpha)[L; (\varphi(l) = a'') : \mathcal{L}]^{\varphi}_{\Phi}$

  The proof of the first assumption must end in a use of rule OVERRIDE. Thus, we have $\Gamma \vdash \zeta(x : \alpha)[L : \mathcal{L}]^{\varphi}_{\Phi} : \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}}$ for some $\mathcal{L}'$, $\mathcal{A}$ and $\mathcal{K}$ such that $\tau = \varsigma(\alpha)[\mathcal{L}']_{\mathcal{K}}^{\mathcal{A}\setminus\{l\}}$. Furthermore, $\Gamma; (\alpha <\# \mathcal{K}); (x : \alpha) \vdash a'' : \mathcal{L}'(l)$. The rule PROTO can be used to prove the first judgment. Hence,

$$\begin{cases} \mathcal{L}' = \mathcal{L} \circ \varphi \\ \mathcal{A} = \operatorname{dom} \mathcal{L}' \setminus \operatorname{dom}(L \circ \varphi) \\ \mathcal{K} = \operatorname{dom} \Phi \\ \operatorname{dom} \mathcal{L} \subseteq \operatorname{dom} L \cup \operatorname{range} \varphi \\ \text{For all } l' \text{ in } \operatorname{dom} L,\ \Gamma; (\alpha <\# \mathcal{K}); (x : \alpha) \vdash L(l') : \mathcal{L}(l') \\ \forall k \in \operatorname{dom} \Phi \cdot (k : \varsigma(\alpha)[\mathcal{L}'']) \in \Gamma \wedge \mathcal{L}'' = \mathcal{L} \circ \Phi(k) \end{cases}$$

Let $L_0 = (L; (\varphi(l) = a''))$, $\mathcal{A}_0 = \mathcal{A} \setminus \{l\}$. We are going to prove that the hypotheses above still hold after substitution of L_0 for L and $\mathcal{A}_0$ for $\mathcal{A}$. This will complete the proof of this case, as it entails $\Gamma \vdash \varsigma(x : \alpha)[L_0 : \mathcal{L}]_{\Phi}^{\varphi} : \varsigma(\alpha)[\mathcal{L}']_{\mathcal{K}}^{\mathcal{A}_0}$, that is, $\Gamma \vdash a' : \tau$.

$$\begin{aligned} \operatorname{dom} \mathcal{L}' \setminus \operatorname{dom}(L_0 \circ \varphi) &= \operatorname{dom} \mathcal{L}' \setminus \operatorname{dom}((L; (\varphi(l) = a'')) \circ \varphi) \\ &= \operatorname{dom} \mathcal{L}' \setminus (\operatorname{dom}(L \circ \varphi) \cup \{l\}) \\ &= (\operatorname{dom} \mathcal{L}' \setminus \operatorname{dom}(L \circ \varphi)) \setminus \{l\}) \\ &= \mathcal{A} \setminus \{l\}) \\ &= \mathcal{A}_0 \end{aligned}$$

$$\begin{aligned} \operatorname{dom} L_0 \cup \operatorname{range} \varphi &= \operatorname{dom}(L; (\varphi(l) = a'')) \cup \operatorname{range} \varphi \\ &= \operatorname{dom} L \cup \{\varphi(l)\} \cup \operatorname{range} \varphi \\ &= \operatorname{dom} L \cup \operatorname{range} \varphi \\ &\supseteq \operatorname{dom} \mathcal{L} \end{aligned}$$

Let $l' \in \operatorname{dom} L_0$. If we assume that $l' \neq \varphi(l)$, then $L_0(l') = L(l')$. Hence, $\Gamma; (\alpha <\# \mathcal{K}); (x : \alpha) \vdash L_0(l') : \mathcal{L}(l')$. Let us now assume $l' = \varphi(l)$. Then $L_0(l') = a''$. But $\Gamma; (\alpha <\# \mathcal{K}); (x : \alpha) \vdash a'' : \mathcal{L}'(l)$ and $\mathcal{L}'(l) = (\mathcal{L} \circ \varphi)(l) = \mathcal{L}(l')$. Hence, again, $\Gamma; (\alpha <\# \mathcal{K}); (x : \alpha) \vdash L_0(l') : \mathcal{L}(l')$.

Other premises are unchanged.

- $\varsigma(x : \alpha)[L : \mathcal{L}]_{\Phi}^{\varphi} \cdot_k l \Leftarrow \varsigma(x : \alpha) a'' \longrightarrow \varsigma(x : \alpha)[L; (\Phi(k)(l) = a'') : \mathcal{L}]_{\Phi}^{\varphi}$

  The proof of the first assumption must end in a use of rule VIEW-OVERRIDE. By inspection, this rule is preceded by subtyping and the rule PROTO. Thus, we have

$$\begin{cases} \tau = \varsigma(\alpha)[\mathcal{L}_1]_{\mathcal{K}'} \\ \Gamma \vdash \varsigma(\alpha)[\mathcal{L}']_{\mathcal{K}}^{\mathcal{A}} \preceq \tau \\ \mathcal{L}' = \mathcal{L} \circ \varphi \\ \mathcal{A} = \operatorname{dom} \mathcal{L}' \setminus \operatorname{dom}(L \circ \varphi) \\ \mathcal{K} = \operatorname{dom} \Phi \\ \operatorname{dom} \mathcal{L} \subseteq \operatorname{dom} L \cup \operatorname{range} \varphi \\ \text{For all } l' \text{ in } \operatorname{dom} L,\ \Gamma; (\alpha <\# \mathcal{K}); (x : \alpha) \vdash L(l') : \mathcal{L}(l') \\ \forall k \in \operatorname{dom} \Phi \cdot (k : \varsigma(\alpha)[\mathcal{L}'']) \in \Gamma \wedge \mathcal{L}'' = \mathcal{L} \circ \Phi(k) \\ \Gamma \vdash \tau <\# \mathcal{K}'' \\ k \in \mathcal{K}'' \\ (k : \varsigma(\alpha)[\mathcal{L}'']) \in \Gamma \\ \Gamma; (\alpha <\# \mathcal{K}''); (x : \alpha) \vdash a'' : \mathcal{L}''(l) \end{cases}$$

Let $L_0 = (L; (\Phi(k)(l) = a))$. We are going to prove that the hypotheses above still hold after substitution of L_0 for L. This will complete the proof of this case, as it entails $\Gamma \vdash \zeta(x : \alpha)[L_0 : \mathcal{L}]^{\varphi}_{\Phi} : \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}}$, and therefore, by subtyping, $\Gamma \vdash a' : \tau$.

By inspection of subtyping rules, we see that we must have $\mathcal{A} = \emptyset$ and $\mathcal{K}' \subseteq \mathcal{K}$.

$$\begin{array}{rcl}
\operatorname{dom} \mathcal{L}' \setminus \operatorname{dom}(L_0 \circ \varphi) & = & \operatorname{dom} \mathcal{L}' \setminus \operatorname{dom}((L; (\Phi(k)(l) = a)) \circ \varphi) \\
& \subseteq & \operatorname{dom} \mathcal{L}' \setminus \operatorname{dom} L \circ \varphi \\
& \subseteq & \mathcal{A} \subseteq \emptyset
\end{array}$$

Therefore, $\operatorname{dom} \mathcal{L}' \setminus \operatorname{dom}(L_0 \circ \varphi) = \emptyset = \mathcal{A}$.

We have $l \in \operatorname{dom} \mathcal{L}'' = \operatorname{dom}(\mathcal{L} \circ \Phi(k))$, and thus, $\Phi(k)(l) \in \operatorname{dom} \mathcal{L}$. Then:

$$\begin{array}{rcl}
\operatorname{dom} L_0 \cup \operatorname{range} \varphi & = & \operatorname{dom}(L; (\Phi(k)(l) = a)) \cup \operatorname{range} \varphi \\
& = & \operatorname{dom} L \cup \{\Phi(k)(l)\} \cup \operatorname{range} \varphi \\
& \supseteq & \operatorname{dom} \mathcal{L} \cup \{\Phi(k)(l)\} \\
& \supseteq & \operatorname{dom} \mathcal{L}
\end{array}$$

Let $l' \in \operatorname{dom} L_0$. If we assume that $l' \neq \Phi(k)(l)$, then we have $L_0(l') = L(l')$. Hence, $\Gamma; (\alpha <\# \mathcal{K}); (x : \alpha) \vdash L_0(l') : \mathcal{L}(l')$. Let us now assume $l' = \Phi(k)(l)$. Then $L_0(l') = a''$. On the other hand, $\Gamma; (\alpha <\# \mathcal{K}''); (x : \alpha) \vdash a'' : \mathcal{L}''(l)$ and $\mathcal{L}''(l) = (\mathcal{L} \circ \Phi(k))(l) = \mathcal{L}(l')$. According to rule Match-Obj, $\mathcal{K}'' \subseteq \mathcal{K}'$. So, $\mathcal{K}'' \subseteq \mathcal{K}$. Hence, again, by lemma 6, $\Gamma; (\alpha <\# \mathcal{K}); (x : \alpha) \vdash L_0(l') : \mathcal{L}(l')$.

Other hypotheses are unchanged.

- $v.l \longrightarrow L(\varphi(l))\{\tau''/\alpha\}\{v/x\}$ where $v = \zeta(x : \alpha)[L : \mathcal{L}]^{\varphi}_{\Phi}$ and $\tau'' = \zeta(\alpha)[\mathcal{L} \circ \varphi]_{\operatorname{dom} \Phi}$

  The proof of the first assumption must end in a use of rule Selection. By inspection, this rule is preceded by subtyping and the rule Proto. Thus, we have

$$\left\{\begin{array}{l}
\tau = (\mathcal{L}_1(l))\{\tau'/\alpha\} \\
\tau' = \zeta(\alpha)[\mathcal{L}_1]_{\mathcal{K}_1} \\
\Gamma \vdash \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}} \preceq \tau' \\
\mathcal{L}' = \mathcal{L} \circ \varphi \\
\mathcal{A} = \operatorname{dom} \mathcal{L}' \setminus \operatorname{dom}(L \circ \varphi) \\
\mathcal{K} = \operatorname{dom} \Phi \\
\operatorname{dom} \mathcal{L} \subseteq \operatorname{dom} L \cup \operatorname{range} \varphi \\
\text{For all } l' \text{ in } \operatorname{dom} L,\ \Gamma; (\alpha <\# \mathcal{K}); (x : \alpha) \vdash L(l') : \mathcal{L}(l') \\
\forall k \in \operatorname{dom} \Phi \cdot (k : \zeta(\alpha)[\mathcal{L}'']) \in \Gamma \wedge \mathcal{L}'' = \mathcal{L} \circ \Phi(k)
\end{array}\right.$$

  By inspection of subtyping rules, we see that $\mathcal{L}_1 \subseteq \mathcal{L}'$. Thus, $\mathcal{L}_1(l) = \mathcal{L}'(l) = \mathcal{L}(\varphi(l))$. We therefore have $\Gamma; (\alpha <\# \mathcal{K}); (x : \alpha) \vdash L(\varphi(l)) : \mathcal{L}_1(l)$. Moreover, $\Gamma \vdash \tau'' <\# \mathcal{K}$. Thus, by lemmas 7, $\Gamma; (x : \tau'') \vdash L(\varphi(l))\{\tau''/\alpha\} : \mathcal{L}_1(l)\{\tau''/\alpha\}$.

  On the other hand, one can see that $\tau'' = \zeta(\alpha)[\mathcal{L}']_{\mathcal{K}}$, and therefore, $\Gamma \vdash v : \tau''$. Using lemma 9, $\Gamma \vdash L(\varphi(l))\{\tau''/\alpha\}\{v/x\} : \mathcal{L}_1(l)\{\tau''/\alpha\}$.

We have $\Gamma \vdash \tau'' \preceq \tau'$. Let us first assume that α appears covariantly in $\mathcal{L}_1$. Then, using lemma 12, $\Gamma \vdash L(\varphi(l))\{\tau''/\alpha\}\{v/x\} : \mathcal{L}_1(l)\{\tau'/\alpha\}$ as desired. Otherwise, by inspection of the subtyping rules, we see that we must have $\tau'' = \tau'$ (no subtyping rule applies). Hence the result.

- $v.{}_k l \longrightarrow L(\Phi(k)(l))\{\tau''/\alpha\}\{v/x\}$ where $v = \zeta(x : \alpha)[L : \mathcal{L}]^{\varphi}_{\Phi}$ and $\tau'' = \zeta(\alpha)[\mathcal{L} \circ \varphi]_{\mathrm{dom}\,\Phi}$

  The proof of the first assumption must end in a use of rule VIEW-SELECTION. By inspection, this rule is preceded by subtyping and the rule PROTO. Thus, we have

$$\begin{cases} \tau = (\mathcal{L}''(l))\{\tau'/\alpha\} \\ \tau' = \zeta(\alpha)[\mathcal{L}_1]_{\mathcal{K}'} \\ \Gamma \vdash \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}} \preceq \tau' \\ \mathcal{L}' = \mathcal{L} \circ \varphi \\ \mathcal{A} = \mathrm{dom}\,\mathcal{L}' \setminus \mathrm{dom}(L \circ \varphi) \\ \mathcal{K} = \mathrm{dom}\,\Phi \\ \mathrm{dom}\,\mathcal{L} \subseteq \mathrm{dom}\,L \cup \mathrm{range}\,\varphi \\ \text{For all } l' \text{ in } \mathrm{dom}\,L,\ \Gamma; (\alpha <\# \mathcal{K}); (x : \alpha) \vdash L(l') : \mathcal{L}(l') \\ \forall k \in \mathrm{dom}\,\Phi \cdot (k : \zeta(\alpha)[\mathcal{L}'']) \in \Gamma \wedge \mathcal{L}'' = \mathcal{L} \circ \Phi(k) \\ \Gamma \vdash \tau' <\# \mathcal{K}'' \\ k \in \mathcal{K}'' \\ (k : \zeta(\alpha)[\mathcal{L}'']) \in \Gamma \end{cases}$$

  By inspection of the subtyping rules, we see that $\mathcal{K}' \subseteq \mathcal{K}$. On the other hand, according to rule MATCH-OBJ, $\mathcal{K}'' \subseteq \mathcal{K}'$. Hence, $k \in \mathcal{K}$ and then $\mathcal{L}''(l) = \mathcal{L}(\Phi(k)(l))$. Therefore, $\Gamma; (\alpha <\# \mathcal{K}); (x : \alpha) \vdash L(\Phi(k)(l)) : \mathcal{L}''(l)$. Moreover, $\Gamma \vdash \tau'' <\# \mathcal{K}$. Thus, by lemma 7, $\Gamma; (x : \tau'') \vdash L(\Phi(k)(l))\{\tau''/\alpha\} : \mathcal{L}''(l)\{\tau''/\alpha\}$.

  On the other hand, one can see that $\tau'' = \zeta(\alpha)[\mathcal{L}']_{\mathcal{K}}$, and therefore, $\Gamma \vdash v : \tau''$. Using lemma 9, $\Gamma \vdash L(\Phi(k)(l))\{\tau''/\alpha\}\{v/x\} : \mathcal{L}''(l)\{\tau''/\alpha\}$.

  We have $\Gamma \vdash \tau'' \preceq \tau'$. Let us first assume that α appears covariantly in $\mathcal{L}''$. Hence, using lemma 12, $\Gamma \vdash L(\Phi(k)(l))\{\tau''/\alpha\}\{v/x\} : \mathcal{L}''(l)\{\tau'/\alpha\}$ as desired. Otherwise, by inspection of the subtyping rules, we see that we must have $\tau'' = \tau'$ (no subtyping rule applies). Hence the result.

- $\langle \zeta(x : \alpha)[L : \mathcal{L}]^{\varphi}_{\Phi} \rangle_k \longrightarrow \zeta(x : \alpha)[L : \mathcal{L}]^{\varphi}_{\Phi;(k=\varphi)}$

  The proof of the first assumption must end in a use of rule VIEW-CAPTURE. Thus, we have $(k : \zeta(\alpha)[\mathcal{L}_1]) \in \Gamma$ and $\Gamma \vdash \zeta(x : \alpha)[L : \mathcal{L}]^{\varphi}_{\Phi} : \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}}$ for some $\mathcal{L}'$, $\mathcal{A}$ and $\mathcal{K}$ such that $\mathcal{L}_1 \subseteq \mathcal{L}'$ and $\tau = \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K} \cup \{k\}}$. The latest judgment can be proved using the rule PROTO. Hence,

$$\begin{cases} \mathcal{L}' = \mathcal{L} \circ \varphi \\ \mathcal{A} = \mathrm{dom}\,\mathcal{L}' \setminus \mathrm{dom}(L \circ \varphi) \\ \mathcal{K} = \mathrm{dom}\,\Phi \\ \mathrm{dom}\,\mathcal{L} \subseteq \mathrm{dom}\,L \cup \mathrm{range}\,\varphi \\ \text{For all } l \text{ in } \mathrm{dom}\,L,\ \Gamma; (\alpha <\# \mathcal{K}); (x : \alpha) \vdash L(l) : \mathcal{L}(l) \\ \forall k \in \mathrm{dom}\,\Phi \cdot (k : \zeta(\alpha)[\mathcal{L}'']) \in \Gamma \wedge \mathcal{L}'' = \mathcal{L} \circ \Phi(k) \end{cases}$$

  Let $\Phi_0 = \Phi; (k = \varphi)$ and $\mathcal{K}_0 = \mathcal{K} \cup \{k\}$. We are going to prove that the hypotheses above still hold after a substitution of Φ_0 for Φ and $\mathcal{K}_0$ for $\mathcal{K}$.

This will complete the proof of this case, as it entails $\Gamma \vdash \zeta(x:\alpha)[L:\mathcal{L}]^{\varphi}_{\Phi_0} : \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}_0}$, that is, $\Gamma \vdash a' : \tau$.

We trivially have $\mathcal{K}_0 = \operatorname{dom} \Phi_0$.

We have $\mathcal{K} \subseteq \mathcal{K}_0$. Therefore, by lemma 6, for all l in $\operatorname{dom} L$, $\Gamma; (\alpha <\# \mathcal{K}_0); (x:\alpha) \vdash L(l) : \mathcal{L}(l)$

Finally, $(k : \zeta(\alpha)[\mathcal{L}_1]) \in \Gamma$ and $\mathcal{L}_1 \subseteq \mathcal{L}' = \mathcal{L} \circ \varphi = \mathcal{L} \circ \Phi_0(k)$.

Other hypotheses are unchanged.

- $\zeta(x:\alpha)[L:\mathcal{L}]^{\varphi}_{\Phi}|_k \longrightarrow \zeta(x:\alpha)[L:\mathcal{L}]^{\Phi(k)}_{\Phi}$

  The proof of the first assumption must end in a use of rule View-Replace. By inspection, this rule is preceded by subtyping and the rule Proto. Thus, we have

$$\begin{cases} \tau = \zeta(\alpha)[\mathcal{L}'']_{\mathcal{K}''} \\ \Gamma \vdash \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}} \preceq \zeta(\alpha)[\mathcal{L}_1]_{\mathcal{K}''} \\ k \in \mathcal{K}'' \\ (k : \zeta(\alpha)[\mathcal{L}'']) \in \Gamma \\ \mathcal{L}' = \mathcal{L} \circ \varphi \\ \mathcal{A} = \operatorname{dom} \mathcal{L}' \setminus \operatorname{dom}(L \circ \varphi) \\ \mathcal{K} = \operatorname{dom} \Phi \\ \operatorname{dom} \mathcal{L} \subseteq \operatorname{dom} L \cup \operatorname{range} \varphi \\ \text{For all } l' \text{ in } \operatorname{dom} L,\ \Gamma; (\alpha <\# \mathcal{K}); (x:\alpha) \vdash L(l') : \mathcal{L}(l') \\ \forall k \in \operatorname{dom} \Phi \cdot (k : \zeta(\alpha)[\mathcal{L}'']) \in \Gamma \wedge \mathcal{L}'' = \mathcal{L} \circ \Phi(k) \end{cases}$$

  Let $\mathcal{L}'_0 = \mathcal{L} \circ \Phi(k)$ and $\varphi_0 = \Phi(k)$. We are going to prove that the last six hypotheses above still hold after a substitution of $\mathcal{L}'_0$ for $\mathcal{L}$ and φ_0 for φ. This way, we will have $\Gamma \vdash \zeta(x:\alpha)[L:\mathcal{L}]^{\varphi_0}_{\Phi} : \zeta(\alpha)[\mathcal{L}'_0]^{\mathcal{A}}_{\mathcal{K}}$ and we will be able to conclude by subtyping, provided that $\Gamma \vdash \zeta(\alpha)[\mathcal{L}'_0]^{\mathcal{A}}_{\mathcal{K}} \preceq \zeta(\alpha)[\mathcal{L}'']_{\mathcal{K}''}$.

  Trivially, $\mathcal{L}'_0 = \mathcal{L} \circ \varphi_0$.

  From the second hypothesis, we deduce that $\mathcal{A} = \emptyset$. Hence, $\operatorname{dom}(\mathcal{L} \circ \varphi) = \operatorname{dom} \mathcal{L}' \subseteq \operatorname{dom}(L \circ \varphi)$. Thus, $\operatorname{dom} \mathcal{L} \cap \operatorname{range} \varphi \subseteq \operatorname{dom} L \cap \operatorname{range} \varphi \subseteq \operatorname{dom} L$. Then, using the hypothesis $\operatorname{dom} \mathcal{L} \subseteq \operatorname{dom} L \cup \operatorname{range} \varphi$, we can deduce $\operatorname{dom} \mathcal{L} \subseteq \operatorname{dom} L$.

  Therefore:

$$\begin{aligned} \operatorname{dom} \mathcal{L}'_0 \setminus \operatorname{dom}(L \circ \varphi_0) &= \operatorname{dom}(\mathcal{L} \circ \Phi(k)) \setminus \operatorname{dom}(L \circ \Phi(k)) \\ &= \emptyset \\ &= \mathcal{A} \end{aligned}$$

  Finally,

$$\begin{aligned} \operatorname{dom} \mathcal{L} &\subseteq \operatorname{dom} L \\ &\subseteq \operatorname{dom} L \cup \operatorname{range} \varphi_0 \end{aligned}$$

  Other hypotheses are unchanged.

  We now prove $\Gamma \vdash \zeta(\alpha)[\mathcal{L}'_0]^{\mathcal{A}}_{\mathcal{K}} \preceq \zeta(\alpha)[\mathcal{L}'']_{\mathcal{K}''}$. From the judgment $\Gamma \vdash \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}} \preceq \zeta(\alpha)[\mathcal{L}_1]_{\mathcal{K}''}$, we deduce that $\mathcal{A} = \emptyset$ and $\mathcal{K}'' \subseteq \mathcal{K}$.

Using the rule Sub-Proto, we are left to prove that $\Gamma \vdash \zeta(\alpha)[\mathcal{L}'_0]_{\mathcal{K}} \preceq \zeta(\alpha)[\mathcal{L}'']_{\mathcal{K}''}$. As $k \in \mathcal{K}'' \subseteq \mathcal{K}$ and $(k : \zeta(\alpha)[\mathcal{L}'']) \in \Gamma$, we have $\mathcal{L}'' = \mathcal{L} \circ \Phi(k) = \mathcal{L}'_0$. If $\mathcal{K} = \mathcal{K}''$, we are done. Otherwise, the judgment $\Gamma \vdash \zeta(\alpha)[\mathcal{L}']_{\mathcal{K}} \preceq \zeta(\alpha)[\mathcal{L}_1]_{\mathcal{K}''}$ an only be proved using the rule Sub-Object. Therefore, $\forall k \in \mathcal{K}'' \cdot (k : \zeta(\alpha)[\mathcal{L}'']) \in \Gamma \wedge \mathrm{co}_\alpha(\mathcal{L}'')$ In particular, α appears covariantly in $\mathcal{L}''$. We can therefore use the rule Sub-Object to prove the desired judgment $\Gamma \vdash \zeta(\alpha)[\mathcal{L}'']_{\mathcal{K}} \preceq \zeta(\alpha)[\mathcal{L}'']_{\mathcal{K}''}$.

- $(\lambda(x : \tau)a)(v) \longrightarrow a\{v/x\}$

  The proof of the first assumption must end in a use of rule App. By inspection, this rule is preceded by subtyping and the rule Abs. Thus, we have
  $$\begin{cases} \Gamma \vdash v : \tau' \\ \Gamma \vdash \tau'_1 \to \tau_1 \preceq \tau' \to \tau \\ \Gamma; (x : \tau'_1) \vdash a : \tau_1 \end{cases}$$
  By inspection of the subtyping rules, we see that the second judgment must be proved using either rule Sub-Refl or Sub-Arrow. In both cases, we have $\Gamma \vdash \tau_1 \preceq \tau$ and $\Gamma \vdash \tau' \preceq \tau'_1$. Then, using rule Sub, we have $\Gamma \vdash v : \tau'_1$. By lemma 9, $\Gamma \vdash a\{v/x\} : \tau_1$. We conclude using rule Sub: $\Gamma \vdash a\{v/x\} : \tau$.

- `open` (`pack` v `as` $\exists(k)\tau$ `hiding` $\mathcal{K}$) `as` $[k, x]$ `in` $a \longrightarrow a\{\mathcal{K}/k\}\{v/x\}$ The proof of the first assumption must end in a use of rule Open. Thus, we have $\Gamma \vdash$ `pack` v `as` $\exists(k)\tau$ `hiding` $\mathcal{K} : \exists(k)\tau'$ and $\Gamma; (k); (x : \tau') \vdash a'' : \tau$ By inspection of the typing rules, we see that the first of these judgments must be proved using the rule Pack. Hence, $\Gamma \vdash v : \tau'\{\mathcal{K}/k\}$.

  By lemma 8, $\Gamma; (x : \tau'\{\mathcal{K}/k\}) \vdash a''\{\mathcal{K}/k\} : \tau$. Then, by lemma 9, $\Gamma \vdash a''\{\mathcal{K}/k\}\{v/x\} : \tau$.

- $F[\rho(k : t)a] \longrightarrow \rho(k : t)F[a]$ where $F \neq [\_]$.

  This case is proved by induction on the size of the context. We only consider the case when $F = \_(a')$. The other cases are similar.

  We therefore show that $(\rho(k : t)a)(a') \longrightarrow \rho(k : t)(a(a'))$.

  We first notice that the rule View-Abs commute with the rule Sub. Indeed, let us assume that we have the following derivation:

  $$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma; (k : t) \vdash a : \tau}}{\Gamma \vdash \rho(k : t)a : \tau}\ (\text{View-Abs}) \qquad \dfrac{\vdots}{\Gamma \vdash \tau \preceq \tau'}}{\Gamma \vdash \rho(k : t)a : \tau'}\ (\text{Sub})$$

  By lemma 5, $\Gamma; (k : t) \vdash \tau \preceq \tau'$. Thus:

  $$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma; (k : t) \vdash a : \tau} \qquad \dfrac{\vdots}{\Gamma; (k : t) \vdash \tau \preceq \tau'}}{\Gamma; (k : t) \vdash a : \tau'}\ (\text{Sub})}{\Gamma \vdash \rho(k : t)a : \tau'}\ (\text{View-Abs})$$

We can therefore assume that the rule View-Abs is not followed by subtyping.

The proof of the first assumption must end in a use of rule Extension, preceded by rule View-Abs:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma;(k:t)\vdash a:\tau'\to\tau}}{\Gamma\vdash\rho(k:t)a:\tau'\to\tau}\ (\text{View-Abs}) \qquad \dfrac{\vdots}{\Gamma\vdash a':\tau'}}{\Gamma\vdash\rho(k:t)a(a'):\tau}\ (\text{App})$$

By lemma 5, $\Gamma;(k:t)\vdash a':\tau'$. Thus:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma;(k:t)\vdash a:\tau'\to\tau} \qquad \dfrac{\vdots}{\Gamma;(k:t)\vdash a':\tau'}}{\Gamma;(k:t)\vdash a(a'):\tau}\ (\text{App})}{\Gamma\vdash\rho(k:t)a(a'):\tau}\ (\text{View-Abs})$$

■

A.2 Progress

Lemma 13 (Type of values) *The values can be classified from their types:*

- *if* $\Gamma\vdash v:\tau'\to\tau$ *then* v *is a function* $\lambda(x:\tau)a$*;*
- *if* $\Gamma\vdash v:t_{\mathcal{K}}$ *or* $\Gamma\vdash v:t_{\mathcal{K}}^{\mathcal{A}}$ *then* v *is an object* $\zeta(x:\alpha)[L:\mathcal{L}]_{\Phi}^{\varphi}$*;*
- *if* $\Gamma\vdash v:\exists(k)\tau$ *then* v *is a view abstraction* $\texttt{pack}\ a\ \texttt{as}\ \tau\ \texttt{hiding}\ \mathcal{K}$*.*

Proof. By inspection of the typing rules, one can see that:

- If v is a function $\lambda(x:a)$, then its type is of the form $\tau'\to\tau$;
- If v is an object $\zeta(x:\alpha)[L:\mathcal{L}]_{\Phi}^{\varphi}$, then its type is of the form $t_{\mathcal{K}}$ or $t_{\mathcal{K}}^{\mathcal{A}}$;
- If v is a view abstraction $\texttt{pack}\ a\ \texttt{as}\ \tau\ \texttt{hiding}\ \mathcal{K}$ then its type is of the form $\exists(k)\tau$.

As we have considered all cases and the form of types are disjoint, we can deduce the form of a value from its type. ■

We can now start the progress proof.

Proof. Let us assume that $\vdash a:\tau$ and that a is not of the form $E[v]$. Let us consider an evaluation context F such that $a=E[F[a_0]]$ and such that a_0 is not a value. According to the lemma 4, $\vdash a_0:\tau'$ for some type τ'. If the lemma is true for a_0 then $a_0\longrightarrow a_0'$ and therefore $a\longrightarrow E[a_0']$. We can thus assume that the grater evaluation context F such that $a=E[F[a_0]]$ and such that a_0 is not a value is the empty context.

On can conclude by considering the different possible form for a. There is no specific difficulty. We therefore only present a typical case. Other cases are proved in a similar way.

- $a = a_1.l \Leftarrow \varsigma(x:\alpha)a'_1$

  a_1 must be a value. Moreover, a derivation of $\vdash a : \tau$ contains the following rule :

$$\frac{\Gamma \vdash a_1 : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}} \qquad \Gamma; (\alpha <\# \mathcal{K}); (x:\alpha) \vdash a'_1 : \mathcal{L}(l)}{\Gamma \vdash a_1.l \Leftarrow \varsigma(x:\alpha)a'_1 : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}\setminus\{l\}}_{\mathcal{K}}}\ (\textsc{Overriding})$$

  The value a_1 has an object type. It is therefore an object $\zeta(x:\alpha)[L:\mathcal{L}_1]^{\varphi}_{\Phi}$.

  Moreover, by inspection of the subtyping rules and of the typing rule Proto, on can see that $\operatorname{dom} L \subseteq \operatorname{dom} \varphi$. As a consequence, $l \in \operatorname{dom}\varphi$, and the reduction rule specifying method overriding can be used, yielding the expression $a' = \zeta(x:\alpha)[L; (\varphi(l) = a'_1) : \mathcal{L}_1]^{\varphi}_{\Phi}$

■

A.3 Translation soundness

Proof. By induction on the translation rules.

- $$\frac{}{\Gamma \vdash \emptyset : (\emptyset, \zeta(\alpha)[\emptyset]^{\emptyset}_{\emptyset}) \Rightarrow \zeta(x:\alpha)[\emptyset:\emptyset]^{\emptyset}_{\emptyset}}\ (\textsc{Empty})$$

  Trivially,

$$\frac{\cdots}{\overline{\Gamma} \vdash \zeta(x:\alpha)[\emptyset:\emptyset]^{\emptyset}_{\emptyset} : \zeta(\alpha)[\emptyset]^{\emptyset}_{\emptyset}}\ (\textsc{Class})$$

- $$\frac{(c : (\mathcal{K}, \tau' \to \tau)) \in \Gamma \qquad \Gamma \vdash a : \tau'_1 \Rightarrow \overline{a} \qquad \Gamma \vdash \tau'_1 \preceq \tau'}{\Gamma \vdash \texttt{inherit}\ c(a) : (\mathcal{K}, \tau) \Rightarrow c(\overline{a})}\ (\textsc{Inheritance})$$

  Then,

$$\frac{\dfrac{(c : \tau' \to \tau) \in \overline{\Gamma}}{\overline{\Gamma} \vdash c : \tau' \to \tau}\ (\textsc{Var}) \qquad \dfrac{\overline{\Gamma} \vdash \overline{a} : \tau'_1 \qquad \overline{\Gamma} \vdash \tau'_1 \preceq \tau'}{\overline{\Gamma} \vdash \overline{a} : \tau'}\ (\textsc{Sub})}{\overline{\Gamma} \vdash c(\overline{a}) : \tau}\ (\textsc{App})$$

- $$\frac{\Gamma \vdash \mathit{extension} : (\mathcal{K}', \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}) \Rightarrow a}{\Gamma \vdash (\mathit{extension}\ \texttt{abstract}\ l : \varsigma(\alpha)\tau) : (\mathcal{K}', \zeta(\alpha)[\mathcal{L}; (l:\tau)]^{\mathcal{A}\cup\{l\}}_{\mathcal{K}}) \Rightarrow a + (l : \varsigma(\alpha)\tau)}\ (\textsc{Abstract-method})$$

  Then,

$$\frac{\overline{\Gamma} \vdash a : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}}{\overline{\Gamma} \vdash a + (l : \varsigma(\alpha)\tau) : \zeta(\alpha)[\mathcal{L}; (l:\tau)]^{\mathcal{A}\cup\{l\}}_{\mathcal{K}}}\ (\textsc{Extension})$$

- $$\frac{\Gamma \vdash \mathit{extension} : (\mathcal{K}', \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}) \Rightarrow a}{\Gamma, (k : \zeta(\alpha)[\mathcal{L}]) \vdash \mathit{extension} : (\mathcal{K}' \cup \{k\}, \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}\cup\{k\}}) \Rightarrow \langle a \rangle_k}\ (\textsc{Local-View})$$

We assume $\overline{\Gamma} \vdash a : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}$. Therefore, by lemma 5, $\overline{\Gamma}; (k : \zeta(\alpha)[\mathcal{L}]) \vdash a : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}$ Then,

$$\frac{\begin{array}{c}\overline{\Gamma}; (k : \zeta(\alpha)[\mathcal{L}]) \vdash a : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}} \\ (k : \zeta(\alpha)[\mathcal{L}]) \in \overline{\Gamma}; (k : \zeta(\alpha)[\mathcal{L}])\end{array}}{\overline{\Gamma}; (k : \zeta(\alpha)[\mathcal{L}]) \vdash \langle a \rangle_k : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K} \cup \{k\}}} \text{ (View-Capture)}$$

- $$\frac{\begin{array}{c}\Gamma, (k : \zeta(\alpha)[\mathcal{L}]) \vdash \mathit{body} : (\mathcal{K}', \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}) \Rightarrow a' \\ \Gamma; (k : \zeta(\alpha)[\mathcal{L}]); (\alpha <\# \mathcal{K}); (x : \alpha) \vdash a : \mathcal{L}(l) \Rightarrow \overline{a}\end{array}}{\begin{array}{c}\Gamma, (k : \zeta(\alpha)[\mathcal{L}]) \vdash \mathit{body}\, \texttt{method}\; l = \varsigma(x : \alpha) a : \\ (\mathcal{K}', \zeta(\alpha)[\mathcal{L}]^{\mathcal{A} \setminus \{l\}}_{\mathcal{K}}) \Rightarrow a'.l \Leftarrow \varsigma(x : \alpha)\overline{a}\end{array}} \text{ (Method)}$$

  Then,

  $$\frac{\begin{array}{c}\overline{\Gamma}; (k : \zeta(\alpha)[\mathcal{L}]) \vdash a' : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}} \\ \overline{\Gamma}; (k : \zeta(\alpha)[\mathcal{L}]); (\alpha <\# \mathcal{K}); (x : \alpha) \vdash \overline{a} : \mathcal{L}(l)\end{array}}{\overline{\Gamma}; (k : \zeta(\alpha)[\mathcal{L}]) \vdash a'.l \Leftarrow \varsigma(x : \alpha)\overline{a} : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A} \setminus \{l\}}_{\mathcal{K}}} \text{ (Override)}$$

- $$\frac{\Gamma, (k : \zeta(\alpha)[\mathcal{L}]) \vdash \mathit{body} : (\mathcal{K}, \tau) \Rightarrow a}{\Gamma, (k : \zeta(\alpha)[\mathcal{L}]) \vdash \texttt{object}\; \mathit{body}\; \texttt{end} : (\mathcal{K}, \tau)\tau \Rightarrow a} \text{ (Class-Body)}$$

  Trivial.

- $$\frac{\begin{array}{c}\Gamma, (k : \zeta(\alpha)[\mathcal{L}]) \vdash \mathit{expr} : (\mathcal{K}', \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}) \Rightarrow a \\ \Gamma \vdash \mathit{type} : \zeta(\alpha)[\mathcal{L}_1]^{\mathcal{A}}_{\mathcal{K}_1} \qquad \mathcal{L}_1 \subseteq \mathcal{L} \\ \mathcal{K}_1 \subseteq \mathcal{K} \qquad \mathcal{A} \subseteq \operatorname{dom} \mathcal{L}_1\end{array}}{\begin{array}{c}\Gamma, (k : \zeta(\alpha)[\mathcal{L}]) \vdash (\mathit{expr} : \mathit{type}) : \\ (\mathcal{K} \setminus \mathcal{K}_1, \zeta(\alpha)[\mathcal{L}_1]^{\mathcal{A}}_{\mathcal{K}}) \Rightarrow a \setminus (\operatorname{dom} \mathcal{L} \setminus \operatorname{dom} \mathcal{L}_1)\end{array}} \text{ (Class-Constraint)}$$

  Then, $\operatorname{dom} \mathcal{L} \setminus \operatorname{dom} \mathcal{L}_1 \subseteq \operatorname{dom} \mathcal{L} \setminus \mathcal{A}$ and $\mathcal{L}|_{\operatorname{dom} \mathcal{L} \setminus (\operatorname{dom} \mathcal{L} \setminus \operatorname{dom} \mathcal{L}_1)} = \mathcal{L}_1$. Therefore,

  $$\frac{\begin{array}{c}\overline{\Gamma}; (k : \zeta(\alpha)[\mathcal{L}]) \vdash a : \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}} \\ \operatorname{dom} \mathcal{L} \setminus \operatorname{dom} \mathcal{L}_1 \subseteq \operatorname{dom} \mathcal{L} \setminus \mathcal{A}\end{array}}{\overline{\Gamma}; (k : \zeta(\alpha)[\mathcal{L}]) \vdash a \setminus (\operatorname{dom} \mathcal{L} \setminus \operatorname{dom} \mathcal{L}_1) : \zeta(\alpha)[\mathcal{L}_1]^{\mathcal{A}}_{\mathcal{K}}} \text{ (Restriction)}$$

- $$\frac{\begin{array}{c}\Gamma; (x : \tau'), (k : \zeta(\alpha)[\mathcal{L}]) \vdash \mathit{expr} : (\mathcal{K}', \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}}) \Rightarrow a' \\ \Gamma; (k); (c : (\{k\}, \tau' \to \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K} \setminus \mathcal{K}' \cup \{k\}})) \vdash a : \tau \Rightarrow \overline{a}\end{array}}{\begin{array}{c}\Gamma \vdash \texttt{class}\; c\, [k](x : \tau') = \mathit{expr}\; \texttt{in}\; a : \tau \Rightarrow \\ \texttt{open}\; (\rho(k : \zeta(\alpha)[\mathcal{L}])\texttt{pack}\; \lambda(x : \tau')a' \\ \texttt{as}\; \exists(k')(\tau' \to \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K} \setminus \mathcal{K}' \cup \{k'\}}) \\ \texttt{hiding}\; \mathcal{K}')\; \texttt{as}\; [k, c]\; \texttt{in}\; \overline{a}\end{array}} \text{ (Private-Class)}$$

We can assume

$$\left\{\begin{array}{l}\overline{\Gamma};(x:\tau');(k:\zeta(\alpha)[\mathcal{L}])\vdash a':\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}}\\ \overline{\Gamma};(k);(c:\tau'\to\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\setminus\mathcal{K}'\cup\{k\}})\vdash\overline{a}:\tau\end{array}\right.$$

Hence,

$$\frac{\overline{\Gamma};(x:\tau');(k:\zeta(\alpha)[\mathcal{L}])\vdash a':\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}}}{\overline{\Gamma};(k:\zeta(\alpha)[\mathcal{L}])\vdash\lambda(x:\tau')a':\tau'\to\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}}}\ (\textsc{Abs})$$

We have $\mathcal{K}'\subseteq\mathcal{K}$. and, therefore, $(\mathcal{K}\setminus\mathcal{K}'\cup\{k'\})\{\mathcal{K}'/k'\}=\mathcal{K}$. Let τ_0 be $\exists(k')(\tau'\to\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\setminus\mathcal{K}'\cup\{k'\}})$. Then,

$$\cfrac{\cfrac{\overline{\Gamma};(k:\zeta(\alpha)[\mathcal{L}])\vdash\lambda(x:\tau')a':\tau'\to\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}}}{\overline{\Gamma};(k:\zeta(\alpha)[\mathcal{L}])\vdash\texttt{pack}\ \lambda(x:\tau')a'\ \texttt{as}\ \tau_0\ \texttt{hiding}\ \mathcal{K}':\tau_0}\ (\textsc{Pack})}{\overline{\Gamma}\vdash\rho(k:\zeta(\alpha)[\mathcal{L}])\texttt{pack}\ \lambda(x:\tau')a'\ \texttt{as}\ \tau_0\ \texttt{hiding}\ \mathcal{K}':\tau_0}\ (\textsc{View-Abs})$$

Finally,

$$\frac{\begin{array}{c}\overline{\Gamma}\vdash\rho(k:\zeta(\alpha)[\mathcal{L}])\texttt{pack}\ \lambda(x:\tau')a'\ \texttt{as}\ \tau_0\ \texttt{hiding}\ \mathcal{K}':\tau_0\\ \overline{\Gamma};(k);(c:\tau'\to\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\setminus\mathcal{K}'\cup\{k\}})\vdash\overline{a}:\tau\end{array}}{\begin{array}{c}\overline{\Gamma}\vdash\texttt{open}\ (\rho(k:\zeta(\alpha)[\mathcal{L}])\texttt{pack}\ \lambda(x:\tau')a'\ \texttt{as}\ \tau_0\\ \texttt{hiding}\ \mathcal{K}')\ \texttt{as}\ [k,c]\ \texttt{in}\ \overline{a}:\tau\end{array}}\ (\textsc{Open})$$

- $$\frac{\begin{array}{c}\Gamma;(x:\tau'),(k:\zeta(\alpha)[\mathcal{L}])\vdash \mathit{expr}:(\mathcal{K}',\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}})\Rightarrow a'\\ \Gamma;(k);(c:(\emptyset,\tau'\to\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\setminus\mathcal{K}'\cup\{k\}}))\vdash a:\tau\Rightarrow\overline{a}\end{array}}{\begin{array}{c}\Gamma\vdash\texttt{class}\ c\ [\texttt{protected}\ k](x:\tau')=\mathit{expr}\ \texttt{in}\ a:\tau\Rightarrow\\ \texttt{open}\ (\rho(k:\zeta(\alpha)[\mathcal{L}])\texttt{pack}\ \lambda(x:\tau')a'\\ \texttt{as}\ \exists(k')(\tau'\to\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\setminus\mathcal{K}'\cup\{k'\}})\\ \texttt{hiding}\ \mathcal{K}')\ \texttt{as}\ [k,c]\ \texttt{in}\ \overline{a}\end{array}}\ (\textsc{Protected-Class})$$

  The proof is exactly the same as in the previous case.

- $$\frac{\begin{array}{c}\Gamma;(x:\tau'),(k:\zeta(\alpha)[\mathcal{L}])\vdash \mathit{expr}:(\mathcal{K}',\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}})\Rightarrow a'\\ \Gamma;(k'');(k':\zeta(\alpha)[\mathcal{L}']);(c:(\emptyset,\tau'\to\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\setminus\mathcal{K}'\cup\{k',k''\}}))\vdash a\{k'/k\}:\tau\Rightarrow\overline{a}\end{array}}{\begin{array}{c}\Gamma\vdash\texttt{class}\ c\ [\texttt{public}\ k](x:\tau')=\mathit{expr}\ \texttt{in}\ a:\tau\Rightarrow\\ \rho(k':\zeta(\alpha)[\mathcal{L}'])\texttt{open}\ (\rho(k:\zeta(\alpha)[\mathcal{L}])\texttt{pack}\ \lambda(x:\tau')\langle a'\rangle_{k'}\\ \texttt{as}\ \exists(k_0)(\tau'\to\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\setminus\mathcal{K}'\cup\{k_0,k'\}})\ \texttt{hiding}\ \mathcal{K}')\ \texttt{as}\ [k'',c]\ \texttt{in}\ \overline{a}\end{array}}\ (\textsc{Public-Class})$$

  We can assume

  $$\left\{\begin{array}{l}\overline{\Gamma};(x:\tau');(k:\zeta(\alpha)[\mathcal{L}])\vdash a':\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}}\\ \overline{\Gamma};(k'');(k':\zeta(\alpha)[\mathcal{L}']);(c:\tau'\to\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\setminus\mathcal{K}'\cup\{k',k''\}}))\vdash\overline{a}:\tau\end{array}\right.$$

  Hence,

  $$\cfrac{\cfrac{\begin{array}{c}\overline{\Gamma};(x:\tau');(k:\zeta(\alpha)[\mathcal{L}])\vdash a:\zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}\\ (k:\zeta(\alpha)[\mathcal{L}])\in\overline{\Gamma};(x:\tau');(k:\zeta(\alpha)[\mathcal{L}])\end{array}}{\overline{\Gamma};(x:\tau');(k:\zeta(\alpha)[\mathcal{L}])\vdash\langle a'\rangle_k:\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\cup\{k\}}}\ (\textsc{Capture})}{\overline{\Gamma};(k:\zeta(\alpha)[\mathcal{L}])\vdash\lambda(x:\tau')\langle a'\rangle_k:\tau'\to\zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\cup\{k\}}}\ (\textsc{Abs})$$

We have $\mathcal{K}' \subseteq \mathcal{K}$. and, therefore, $(\mathcal{K} \setminus \mathcal{K}' \cup \{k_0, k'\})\{\mathcal{K}'/k_0\} = \mathcal{K} \cup \{k'\}$. Let τ_0 be $\exists(k_0)(\tau' \to \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\setminus\mathcal{K}'\cup\{k_0,k'\}})$. Then,

$$\dfrac{\dfrac{\overline{\Gamma}; (k : \zeta(\alpha)[\mathcal{L}]) \vdash \lambda(x : \tau')\langle a' \rangle_k : \tau' \to \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\cup\{k\}}}{\overline{\Gamma}; (k : \zeta(\alpha)[\mathcal{L}]) \vdash \texttt{pack}\ \lambda(x : \tau')\langle a' \rangle_k\ \texttt{as}\ \tau_0\ \texttt{hiding}\ \mathcal{K}' : \tau_0}\ (\textsc{Pack})}{\overline{\Gamma} \vdash \rho(k : \zeta(\alpha)[\mathcal{L}])\texttt{pack}\ \lambda(x : \tau')\langle a' \rangle_k\ \texttt{as}\ \tau_0\ \texttt{hiding}\ \mathcal{K}' : \tau_0}\ (\textsc{View-Abs})$$

By lemma 5,

$$\overline{\Gamma}; (k' : \zeta(\alpha)[\mathcal{L}']) \vdash \rho(k : \zeta(\alpha)[\mathcal{L}])\texttt{pack}\ \lambda(x : \tau')\langle a' \rangle_k\ \texttt{as}\ \tau_0\ \texttt{hiding}\ \mathcal{K}' : \tau_0$$

Therefore, using rule Open:

$$\dfrac{\begin{array}{c}\overline{\Gamma}; (k' : \zeta(\alpha)[\mathcal{L}']) \vdash \rho(k : \zeta(\alpha)[\mathcal{L}])\texttt{pack}\ \lambda(x : \tau')a'\ \texttt{as}\ \tau_0\ \texttt{hiding}\ \mathcal{K}' : \tau_0 \\ \overline{\Gamma}; (k' : \zeta(\alpha)[\mathcal{L}']); (k''); (c : \tau' \to \zeta(\alpha)[\mathcal{L}']^{\mathcal{A}}_{\mathcal{K}\setminus\mathcal{K}'\cup\{k',k''\}}) \vdash \overline{a} : \tau\end{array}}{\begin{array}{c}\overline{\Gamma}; (k' : \zeta(\alpha)[\mathcal{L}']) \vdash \texttt{open}\ (\rho(k : \zeta(\alpha)[\mathcal{L}])\texttt{pack}\ \lambda(x : \tau')a'\ \texttt{as}\ \tau_0 \\ \texttt{hiding}\ \mathcal{K}')\ \texttt{as}\ [k'', c]\ \texttt{in}\ \overline{a} : \tau\end{array}}$$

Finally,

$$\dfrac{\begin{array}{c}\overline{\Gamma}; (k' : \zeta(\alpha)[\mathcal{L}']) \vdash \texttt{open}\ (\rho(k : \zeta(\alpha)[\mathcal{L}])\texttt{pack}\ \lambda(x : \tau')a'\ \texttt{as}\ \tau_0 \\ \texttt{hiding}\ \mathcal{K}')\ \texttt{as}\ [k'', c]\ \texttt{in}\ \overline{a} : \tau\end{array}}{\begin{array}{c}\overline{\Gamma} \vdash \rho(k' : \zeta(\alpha)[\mathcal{L}'])\texttt{open}\ (\rho(k : \zeta(\alpha)[\mathcal{L}])\texttt{pack}\ \lambda(x : \tau')a'\ \texttt{as}\ \tau_0 \\ \texttt{hiding}\ \mathcal{K}')\ \texttt{as}\ [k'', c]\ \texttt{in}\ \overline{a} : \tau\end{array}}\ (\textsc{Vue-Abs})$$

•

$$\dfrac{\begin{array}{c}(c : (\mathcal{K}', \tau' \to \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}})) \in \Gamma \\ \Gamma \vdash \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}} \preceq \zeta(\alpha)[\mathcal{L}]_{\mathcal{K}\setminus\mathcal{K}'}\end{array}}{\Gamma \vdash \texttt{new}\ c : \tau' \to \zeta(\alpha)[\mathcal{L}]_{\mathcal{K}\setminus\mathcal{K}'} \Rightarrow c}\ (\textsc{New})$$

Then,

$$\dfrac{(c : \tau' \to \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}) \in \overline{\Gamma}}{\overline{\Gamma} \vdash c : \tau' \to \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}}}\ (\textsc{Var})$$

and

$$\dfrac{\overline{\Gamma} \vdash \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}} \preceq \zeta(\alpha)[\mathcal{L}]_{\mathcal{K}\setminus\mathcal{K}'} \qquad \dfrac{}{\overline{\Gamma} \vdash \tau' \preceq \tau'}\ (\textsc{Sub-Refl})}{\overline{\Gamma} \vdash \tau' \to \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}} \preceq \tau' \to \zeta(\alpha)[\mathcal{L}]_{\mathcal{K}\setminus\mathcal{K}'}}\ (\textsc{Sub-Arrow})$$

Hence,

$$\dfrac{\begin{array}{c}\overline{\Gamma} \vdash c : \tau' \to \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}} \\ \overline{\Gamma} \vdash \tau' \to \zeta(\alpha)[\mathcal{L}]^{\mathcal{A}}_{\mathcal{K}} \preceq \tau' \to \zeta(\alpha)[\mathcal{L}]_{\mathcal{K}\setminus\mathcal{K}'}\end{array}}{\overline{\Gamma} \vdash c : \tau' \to \zeta(\alpha)[\mathcal{L}]_{\mathcal{K}\setminus\mathcal{K}'}}\ (\textsc{Sub})$$

■