Interpretations of Extensible Objects and Types

Viviana Bono
School of Computer Science
The University of Birmingham
Edgbaston, Birmingham, B152 TT, UK
`V.Bono@cs.bham.ac.uk`

Michele Bugliesi
Dipartimento di Informatica
Università "Ca' Foscari" di Venezia
Via Torino 155, I-30173 Mestre, Italy
`michele@dsi.unive.it`

Abstract. We present a type-theoretic encoding of extensible objects and types. The ambient theory is a higher-order λ-calculus with polymorphic types, recursive types and operators, and subtyping. Using this theory, we give a type preserving and computationally adequate translation of a full-fledged object calculus that includes object extension and override. The translation specializes to calculi of nonextensible objects and validates the expected subtyping relationships.

1 Introduction

The attempt to reduce object-oriented programming to procedural or functional programming is motivated by the desire to give sound and formal foundations to object-oriented languages and their specific constructs and techniques. The research in this area initiated with Cook's work [Coo87,Coo89] on the *generator model*, and Kamin's *self-application semantics* [Kam88]. Refined formulations of the generator model were later proposed by Bruce [Bru94] to give interpretations of *class-based* object calculi. A number of encodings for *object-based* calculi have then been formulated by Pierce and Turner [PT94], Abadi, Cardelli and Viswanathan [AC96,ACV96,Vis98], Bruce, Pierce and Cardelli [BCP97], and by Crary [Cra98]. These interpretations apply to a rich variety of object calculi with primitives of object formation, message send and (functional) method override: they succeed in validating the operational semantics of these calculi as well as the expected subtyping relations.

None of these proposals, however, scales to calculi of extensible objects, where primitives are provided for modifying the size of an object with the addition of new methods. Method addition poses two major problems: the first is the need for `MyType` polymorphic typing of methods, to allow method types to be specialized when methods are inherited; the second arises from the combination of subtyping and object extension [FM95].

The interpretation we present in this paper addresses both these problems. Our source calculus features extensible objects in the spirit of the *Lambda Calculus of Objects* [FHM94] and subsequent calculi [FM95,BL95,BB98]. `MyType` polymorphism is rendered via *match*-bounded polymorphism, as in the system we developed in [BB98]. Subtyping, is accounted for by distinguishing extensible from nonextensible objects as suggested by Fisher and Mitchell in [FM95].

As in other papers on encodings, our interpretation is a translation of the source object calculus into a polymorphic λ-calculus with recursive types and (higher-order) subtyping. In the encoding, extensible objects are represented as recursive records that include "selectable" methods, "method updaters" invoked upon override, as well as "method generators" that reinstall selectable methods upon extension. The contributions of our approach can be summarized as follows.

Firstly, it constitutes the first[1] interpretation of extensible objects into a fully formal functional calculus. The interpretation is faithful to the source calculus, as it is computationally adequate and validates the typing of terms.

Secondly, the translation specializes to the case of nonextensible objects, validating the expected subtypings: although we focus on one particular calculus – specifically, on one approach to combining object extension with subtyping – the translation is general enough to capture other notions of subtyping over object types (notably, the notions of covariant and invariant subtyping of [AC96]).

The rest of the paper is organized as follows. In Sections 2 and 3 we review the object and functional calculi used in the translation. In Sections 4 and 5 we describe the translation of extensible objects. In Section 6 we discuss the interpretation of nonextensible objects and various forms of subtyping relationships. In Section 7 we discuss related work and some final remarks.

2 $\texttt{Ob}^+$: Extensible Objects and Types

The source calculus of our translation, called $\texttt{Ob}^+$, is essentially a typed version of the *Lambda Calculus of Objects* of [FHM94]. There are two differences from the original proposal of [FHM94]: (*i*) the syntax of $\texttt{Ob}^+$ is typed, and (*ii*) methods are ς-abstractions instead of the λ-abstractions of [FHM94]. The typed syntax is useful in the translation, as it ensures that well-typed objects have unique types. The choice of ς-binders makes the syntax of $\texttt{Ob}^+$ a proper extension of the the typed ς-calculus of [AC96], and thus it facilitates comparisons with previous translations in the literature.

Types and Terms. An object type has the form $\texttt{pro}(\texttt{X})\langle\!\langle m_i{:}B_i\{\texttt{X}\}^{i\in[1..n]}\rangle\!\rangle$: it denotes the collection of objects with methods $m_1,\ldots,m_n$ that, when invoked, return values of types $B_1,\ldots,B_n$, respectively, with every free occurrence of $\texttt{X}$ substituted by the $\texttt{pro}$-type itself. Types include type variables, denoted by $\texttt{X}$, $\texttt{U}$, ... The syntax of terms is defined by the following productions:

$a, b ::= x,$	variable
$\varsigma(\texttt{U}, A)\langle m_i = \varsigma(x:\texttt{U})b_i\rangle^{i\in[1..n]}$	object (m_i distinct)
$a \longleftarrow\!+\, m{=}\varsigma(\texttt{U}, A)\varsigma(x:\texttt{U})b$	object extension
$a \leftarrow m{=}\varsigma(\texttt{U}, A)\varsigma(x:\texttt{U})b$	method override
$a \Leftarrow m$	method invocation

An object is a collection of labelled methods: each method has a bound variable that represents self, and a body. In the above productions, the type A is the type of the object, and the type variable $\texttt{U}$ is $\texttt{MyType}$, the type of self. This format

[1] But see [BDZ99] in these proceedings, for a similar approach.

of terms is inspired by [Rem97] and [Liq97]. Unlike those proposals, however, we use two operators for overriding and extension: this choice is well motivated, as the two operations are distinguished by our interpretation. The construct for extension allows the addition of a single method: a simple generalization of the syntax (and of the typing rules) would allow multiple simultaneous additions. The relation of top-level reduction (cf. App. A) extends the reduction relation of [AC96], with a clause for method additions (this clause simplifies the corresponding clause used in [Rem97]). The reflexive and transitive congruence generated by reduction is denoted by $\xrightarrow{obj}$; results are terms in object form (cf. App. A). We say that a closed term a converges – written $a \Downarrow_{obj}$ – if there exists a result v such that $a \xrightarrow{obj} v$.

Type System. The type system of $\mathtt{Ob}^+$ relies on the same form of (implicit) match-bounded polymorphism we studied in [BB98] for the Lambda Calculus of Objects [FHM94]. The typing rules (cf. App A) generalize the corresponding typing rules of [AC96] for nonextensible objects. (Val Extend) requires the object a being extended to be a `pro`-type: method addition is thus typed with *exact* knowledge of the type of a. (Val Send) and (Val Override), instead, are both *structural*, in the sense of [ACV96]. In both rules, the type A may either unknown (i.e. a type variable), or a `pro`-type. When A is a `pro`-type, the operation (invocation or override) is *external*; when it is a type variable, the operation is *self-inflicted*: in both cases, A, (hence the object a), is required to have a method m with type B. In (Val Override), the typing of the method ensures that the new body has the same type as the original method: the bound for the type variable $\mathtt{U}$, denoted by $\Gamma\langle A\rangle$, is either A, if A is a `pro`-type, or the current bound for A declared in the context Γ.

3 The Functional Calculus $\mathtt{F}_{\omega<:\mu}$

The target calculus of the translation is $\mathtt{F}_{\omega<:\mu}$, a variant of the omega-order polymorphic λ-calculus $F^{\omega}_{<:}$ with (higher-order) subtyping, extended with recursive types and operators, recursive functions and records, and local definitions. Types and type operators are collectively called *constructors*. A type operator is a function from types to types. The notation $\mathbb{A} :: \mathcal{K}$ indicates that the constructor $\mathbb{A}$ has kind $\mathcal{K}$, where $\mathcal{K}$ is either $\mathcal{T}$, the kind of types, or $\mathcal{K} \Rightarrow \mathcal{K}$, the kind of type operators. The typing rules are standard (see [AC96], Chap. 20). The following notation is used throughout: *Op* stands for the kind $\mathcal{T} \Rightarrow \mathcal{T}$; $\mathbb{A} \leq \mathbb{B}$ denotes subtyping over type operators; if $\mathbb{A}$ is a constructor of kind *Op*, $\mathbb{A}^*$ denotes the fixed point $\mu(\mathtt{X})\mathbb{A}(\mathtt{X})$ of $\mathbb{A}$; dually, for $\mathbb{A} :: \mathcal{T} \equiv \mu(\mathtt{X})\mathbb{B}(\mathtt{X})$, $\mathbb{A}^{\mathrm{OP}}$ is the type operator $\lambda(\mathtt{X})\mathbb{B}(\mathtt{X}) :: Op$ corresponding to $\mathbb{A}$. The syntax of types and terms, and the reduction rules for $\mathtt{F}_{\omega<:\mu}$ are standard (cf. App. B). Evaluation, denoted by $\xrightarrow{fun}$, is the transitive and reflexive congruence generated by reduction; results include λ-abstractions and records. We say that a closed term a converges – written $a \Downarrow_{fun} v$ – if there exists a result v such that $a \xrightarrow{fun} v$.

4 Overview of the Translation

Looking at the typing rules of $\mathtt{Ob}^{+}$, we may identify two distinguished views of methods: the internal view, in which methods are concrete values, and the external view where methods may be seen as "abstract services" that can be accessed via message sends. The polymorphic typing of methods reflects the internal view, while the external view is provided by the types of methods in the object types. Based on this observation, our translation splits methods into two parts, in ways similar to, but different from, the translation of [ACV96]. Each method m_i is represented by two components: m_i^{poly}, associated with the actual method body, and m_i^{sel} which is selected by a message send.

Given $A \equiv \mathtt{pro}(\mathtt{X})\langle\!\langle m_i : B_i\{\mathtt{X}\}\rangle\!\rangle^{i\in[1..n]}$, the m_i^{sel} components are collected in the *abstract interface* associated with A, which is represented by the type operator $\mathbb{A}^{\text{IN}} \equiv \lambda(\mathtt{X})[m_i^{sel} : \mathbb{B}_i\{\mathtt{X}\}]^{i\in[1..n]}$ (here, and below, $\mathbb{B}_i$ is the translation of B_i). The type A, instead, is represented as the recursive record type $\mathbb{A} = \mu(\mathtt{X})[m_i^{poly} : \forall(\mathtt{U} \le \mathbb{A}^{\text{IN}})\mathtt{U}^{*}\to\mathbb{B}_i\{\mathtt{U}^{*}\},\ m_i^{sel} : \mathbb{B}_i\{\mathtt{X}\}]^{i\in[1..n]}$. Note that the polymorphic components are exposed in the type, as they will be needed in the interpretation of object extension. The translation of objects parallels this interpretation of object types. Letting $\mathbb{A}^{\text{OP}} \equiv \lambda(\mathtt{X})[m_i^{poly} : \forall(\mathtt{U} \le \mathbb{A}^{\text{IN}})\mathtt{U}^{*}\to\mathbb{B}_i\{\mathtt{U}^{*}\},\ m_i^{sel} : \mathbb{B}_i\{\mathtt{X}\}]^{i\in[1..n]}$, the translation of an object $\varsigma(\mathtt{X}, A)\langle m_i = \varsigma(x : \mathtt{X})b_i\rangle^{i\in[1..n]}$ is the recursive record satisfying the equation $a = [m_i^{poly} = \Lambda(\mathtt{U} \le \mathbb{A}^{\text{IN}})\lambda(x : \mathtt{U}^{*})\,[\![\, b_i \,]\!]\,, m_i^{sel} = a.m_i^{poly}(\mathbb{A}^{\text{OP}})(a)]^{i\in[1..n]}$, where $[\![\, b_i \,]\!]$ is the translation of the body b_i. Method bodies, labelled by the m_i^{poly}'s, are represented as polymorphic functions of the self parameter, whose type is $\mathtt{U}^{*}$, the fixed point of the type operator $\mathtt{U}$. The constraint $\mathtt{U} \le \mathbb{A}^{\text{IN}}$ ensures that $\mathtt{U}^{*}$ contains all the m_i^{sel}'s, thus allowing each method to invoke its sibling methods via self. The m_i^{sel} components, in turn, are formed by self-application: method invocation for each m_i may then safely be interpreted as record selection on m_i^{sel}.

Method Override. Method override is accounted for by extending the interpretation of objects with a collection of *updaters*, as in [ACV96]. In the new translation, each method m_i is split in three parts, introducing the updater m_i^{upd}. The function of the updater is to take the method body supplied in the override and return a new object with the new body installed in place of the original: overriding m_i is thus translated by a simple call to m_i^{upd}. The typing of updaters requires a different, and more complex definition of the abstract interface. The problem arises from self-inflicted overrides: if a self-inflicted override is to be translated as a call to the updater, the updater itself must be exposed in the interface $\mathbb{A}^{\text{IN}}$ used in the type of the polymorphic components. But then, since the polymorphic components and the updaters must be typed consistently, the updaters must be exposed in the interface $\mathbb{A}^{\text{IN}}$ used in the type of the updaters themselves. This leads to a definition of the interface as the type operator that satisfies the equation $\mathbb{A}^{\text{IN}} = \lambda(\mathtt{X})[m_i^{upd} : (\forall(\mathtt{U} \le \mathbb{A}^{\text{IN}})\mathtt{U}^{*}\to\mathbb{B}_i\{\mathtt{U}^{*}\})\to\mathtt{X},\ m_i^{sel} : \mathbb{B}_i\{\mathtt{X}\}]$.

5 The Translation, Formally

The translation is given parametrically on contexts. Parameterization on contexts is required to ensure a well-defined translation of type variables.

Table 1: Translation of Types

$$A \equiv \mathtt{pro}(\mathtt{X})\langle m_i : B_i\{\mathtt{X}\}\rangle^{i\in[1..n]}$$

$$
\begin{aligned}
[\![\Gamma', \mathtt{X} \lessdot\!\# A, \Gamma'' \rhd \mathtt{X}]\!]^{\mathrm{IN}} &\triangleq \mathtt{X} \\
[\![\Gamma \rhd A]\!]^{\mathrm{IN}} &\triangleq \mu(\mathtt{Y})\lambda(\mathtt{X})[\, m_i^{upd} : (\forall(\mathtt{U} \le \mathtt{Y})\mathtt{U}^* \to [\![\Gamma, \mathtt{X} \rhd B_i\{\mathtt{X}\}]\!]^{\mathrm{TY}}\{\mathtt{X}{:=}\mathtt{U}^*\}) \to \mathtt{X}, \\
&\qquad\qquad m_i^{sel} : [\![\Gamma, \mathtt{X} \rhd B_i\{\mathtt{X}\}]\!]^{\mathrm{TY}}]^{i\in[1..n]}) \\
[\![\Gamma', \mathtt{X} \lessdot\!\# A, \Gamma'' \rhd \mathtt{X}]\!]^{\mathrm{OP}} &\triangleq \mathtt{X} \\
[\![\Gamma \rhd A]\!]^{\mathrm{OP}} &\triangleq \lambda(\mathtt{X})[\, ext : \forall(\mathtt{U} \le [\![\Gamma \rhd A]\!]^{\mathrm{IN}})\mathtt{U}^* \to \mathtt{U}^* \\
&\qquad m_i^{poly} : \forall(\mathtt{U} \le [\![\Gamma \rhd A]\!]^{\mathrm{IN}})\mathtt{U}^* \to [\![\Gamma, \mathtt{X} \rhd B_i\{\mathtt{X}\}]\!]^{\mathrm{TY}}\{\mathtt{X}{:=}\mathtt{U}^*\}, \\
&\qquad m_i^{upd} : (\forall(\mathtt{U} \le [\![\Gamma \rhd A]\!]^{\mathrm{IN}})\mathtt{U}^* \to [\![\Gamma, \mathtt{X} \rhd B_i\{\mathtt{X}\}]\!]^{\mathrm{TY}}\{\mathtt{X}{:=}\mathtt{U}^*\}) \to \mathtt{X}, \\
&\qquad m_i^{sel} : [\![\Gamma, \mathtt{X} \rhd B_i\{\mathtt{X}\}]\!]^{\mathrm{TY}}]^{i\in[1..n]} \\
[\![\Gamma', \mathtt{X}, \Gamma'' \rhd \mathtt{X}]\!]^{\mathrm{TY}} &\triangleq \mathtt{X} \\
[\![\Gamma', \mathtt{X} \lessdot\!\# A, \Gamma'' \rhd \mathtt{X}]\!]^{\mathrm{TY}} &\triangleq \mathtt{X}^* \\
[\![\Gamma \rhd A]\!]^{\mathrm{TY}} &\triangleq \mu(\mathtt{X})[\, ext : \forall(\mathtt{U} \le [\![\Gamma \rhd A]\!]^{\mathrm{IN}})\mathtt{U}^* \to \mathtt{U}^* \\
&\qquad m_i^{poly} : \forall(\mathtt{U} \le [\![\Gamma \rhd A]\!]^{\mathrm{IN}})\mathtt{U}^* \to [\![\Gamma, \mathtt{X} \rhd B_i\{\mathtt{X}\}]\!]^{\mathrm{TY}}\{\mathtt{X}{:=}\mathtt{U}^*\}, \\
&\qquad m_i^{upd} : (\forall(\mathtt{U} \le [\![\Gamma \rhd A]\!]^{\mathrm{IN}})\mathtt{U}^* \to [\![\Gamma, \mathtt{X} \rhd B_i\{\mathtt{X}\}]\!]^{\mathrm{TY}}\{\mathtt{X}{:=}\mathtt{U}^*\}) \to \mathtt{X}, \\
&\qquad m_i^{sel} : [\![\Gamma, \mathtt{X} \rhd B_i\{\mathtt{X}\}]\!]^{\mathrm{TY}}]^{i\in[1..n]}
\end{aligned}
$$

The translation of types is by structural induction. As in [AC95], the treatment of object types depends on the context where they are used: in certain contexts they are interpreted as type operators, while in other contexts they are interpreted as types. From the translation of contexts and judgments (cf. Table 3), we see that $[\![\cdot]\!]^{\mathrm{IN}}$ and $[\![\cdot]\!]^{\mathrm{TY}}$ are used, respectively, in typing statements of the form $a : A$, and matching statements of the form $A \lessdot\!\# B$. The translation $[\![\cdot]\!]^{\mathrm{OP}}$ is used in the translation of terms in Table 2 below, which also explains the presence of the *ext* field in $[\![\cdot]\!]^{\mathrm{TY}}$ and $[\![\cdot]\!]^{\mathrm{OP}}$.

For the translation of terms, we first introduce a recursive function that forms the (recursive fold of) the record with the m_i^{poly}, m_i^{sel} and m_i^{upd} components, together with the *ext* field needed to encode object extension. There is one such function for each type object type A.

```
$\mathtt{letrec}\ mkobj_A(f_i : \forall(\mathtt{U} \le [\![ \Gamma \rhd A ]\!]^{\mathrm{IN}})\mathtt{U}^* \to \mathbb{B}_i\{\mathtt{U}^*\}^{i\in[1..n]}) : [\![ \Gamma \rhd A ]\!]^{\mathrm{TY}} =$
    $\mathtt{let}\ \mathrm{SELF} : [\![ \Gamma \rhd A ]\!]^{\mathrm{TY}} = mkobj_A(f_1)\dots(f_n)\ \mathtt{in}$
    $\mathtt{fold}([\![ \Gamma \rhd A ]\!]^{\mathrm{TY}},$
        $[ext = \Lambda(\mathtt{U} \le [\![ \Gamma \rhd A ]\!]^{\mathrm{IN}})\lambda(x : \mathtt{U}^*)x$
        $m_i^{poly} = f_i,$
        $m_i^{upd} = \lambda(g : \forall(\mathtt{U} \le [\![ \Gamma \rhd A ]\!]^{\mathrm{IN}})\mathtt{U}^* \to \mathbb{B}_i\{\mathtt{U}^*\})mkobj_A(f_1)\dots(g)\dots(f_n),$
        $m_i^{sel} = \mathtt{unfold}(\mathrm{SELF}).m_i^{poly}([\![ \Gamma \rhd A ]\!]^{\mathrm{OP}})(\mathrm{SELF})]^{i\in[1..n]})$
$\text{where } A \equiv \mathtt{pro}(\mathtt{X})\langle\!\langle m_i : B_i\{\mathtt{X}\}\rangle\!\rangle^{i\in[1..n]}, \text{ and } \mathbb{B}_i\{\mathtt{U}^*\} \equiv [\![ \Gamma, \mathtt{X} \rhd B_i\{\mathtt{X}\} ]\!]^{\mathrm{TY}}\{\mathtt{X}{:=}\mathtt{U}^*\}.$
```

Table 2: Translation of Terms

$$\llbracket\, \Gamma \triangleright {\large\varsigma}(\mathtt{U}, A)\langle m_i = \varsigma(x : \mathtt{U})b_i\rangle^{i\in[1..n]} \,\rrbracket \;\triangleq$$
$$mkobj_A(\Lambda(\mathtt{U} \le \llbracket\, \Gamma \triangleright A \,\rrbracket^{\mathrm{IN}})\lambda(s : \mathtt{U}^*)\, \llbracket\, \Gamma, \mathtt{U} \mathrel{\lessdot\#} A, s : \mathtt{U} \triangleright b_i \,\rrbracket^{\,i\in[1..n]})$$
$$\text{where } A \equiv \mathtt{pro}(\mathtt{X})\langle m_i : B_i\{\mathtt{X}\}\rangle^{i\in[1..n]}$$

$$\llbracket\, \Gamma \triangleright a \longleftarrow\!\!+\; m_{n+1} = {\large\varsigma}(\mathtt{U}, A^+)\varsigma(x : \mathtt{U})b \,\rrbracket \;\triangleq$$
$$\overline{a}.ext\ (\llbracket\, \Gamma \triangleright A \,\rrbracket^{\mathrm{OP}})\ (mkobj_{A^+}(\overline{a}.m_1{}^{poly})\cdots(\overline{a}.m_n{}^{poly})\ (\Lambda(\mathtt{U} \le \llbracket\, \Gamma \triangleright A^+ \,\rrbracket^{\mathrm{IN}})\overline{b}))$$
$$\text{where } A \equiv \mathtt{pro}(\mathtt{X})\langle m_i : B_i\{\mathtt{X}\}\rangle^{i\in[1..n]},\ A^+ \equiv \mathtt{pro}(\mathtt{X})\langle m_i : B_i\{\mathtt{X}\}\rangle^{i\in[1..n+1]},$$
$$\overline{a} \equiv \llbracket\, \Gamma \triangleright a \,\rrbracket,\ \text{and } \overline{b} \equiv \lambda(x{:}\mathtt{U}^*)\, \llbracket\, \Gamma, \mathtt{U} \mathrel{\lessdot\#} A^+, x{:}\mathtt{U} \triangleright b \,\rrbracket$$

$$\llbracket\, \Gamma \triangleright a \leftarrow m = {\large\varsigma}(\mathtt{U}, A)\varsigma(x : \mathtt{U})b \,\rrbracket \;\triangleq$$
$$\mathtt{unfold}(\llbracket\, \Gamma \triangleright a \,\rrbracket).m^{upd}(\Lambda(\mathtt{U} \le \llbracket\, \Gamma \triangleright \Gamma\langle A\rangle \,\rrbracket^{\mathrm{IN}})\lambda(x : \mathtt{U}^*)\ \llbracket\, \Gamma, \mathtt{U} \mathrel{\lessdot\#} \Gamma\langle A\rangle, x{:}\mathtt{U} \triangleright b \,\rrbracket)$$

$$\llbracket\, \Gamma \triangleright a \Leftarrow m \,\rrbracket \;\triangleq\; \mathtt{unfold}(\llbracket\, \Gamma \triangleright a \,\rrbracket).m^{sel}$$

In the clause for object formation, the typing of the m_i^{sel} components requires the relation $\llbracket\, \Gamma \triangleright A \,\rrbracket^{\mathrm{OP}} <: \llbracket\, \Gamma \triangleright A \,\rrbracket^{\mathrm{IN}}$, which is derived by first unrolling the fixed-point, and then applying the rules for constructor subtyping.

A method addition forms a new object by applying $mkobj_{A^+}$ (A^+ is the type of the extended object) to the (translation of) the method bodies of the original object a, and to the newly added method. Selecting the ext field from $\overline{a}$, – the object being extended – guarantees that $\overline{a}$ is evaluated prior to the extension: this is required for computational adequacy as the reduction rules of $\mathtt{Ob}^+$ do require a to be in object form prior to reducing a method addition. The call to $mkobj_{A^+}$ is well typed, as every $m_i^{poly} : \forall(\mathtt{U} \le \mathbb{A}^{\mathrm{IN}})\mathtt{U}^* \to \mathbb{B}_i\{\mathtt{U}^*\}$ may be given, by subsumption, the type $\forall(\mathtt{U} \le (\mathbb{A}^+)^{\mathrm{IN}})\mathtt{U}^* \to \mathbb{B}_i\{\mathtt{U}^*\}$, using $\llbracket\, \Gamma \triangleright \mathbb{A}^+ \,\rrbracket^{\mathrm{IN}} \le \llbracket\, \Gamma \triangleright \mathbb{A} \,\rrbracket^{\mathrm{IN}}$, which holds as $\llbracket\, \Gamma \triangleright \mathbb{A} \,\rrbracket^{\mathrm{IN}}$ is covariant in the bound variable $\mathtt{Y}$.

The translation of method invocation and override on a method m are translated by a call to the corresponding components, m^{sel} or m^{upd}. In both cases, a recursive unfold is required prior to accessing the desired component.

The translation of contexts and judgments is obtained directly from the translation of types and terms.

Table 3: Translation of Contexts and Judgments

$$\llbracket\, \Gamma \vdash * \,\rrbracket \triangleq \llbracket\, \Gamma \,\rrbracket \vdash \diamond \qquad \llbracket\, \Gamma \vdash A \mathrel{\lessdot\#} B \,\rrbracket \triangleq \llbracket\, \Gamma \,\rrbracket \vdash \llbracket\, \Gamma \triangleright A \,\rrbracket^{\mathrm{IN}} \le \llbracket\, \Gamma \triangleright B \,\rrbracket^{\mathrm{IN}}$$
$$\llbracket\, \Gamma \vdash A \,\rrbracket \triangleq \llbracket\, \Gamma \,\rrbracket \vdash \llbracket\, \Gamma \triangleright A \,\rrbracket^{\mathrm{TY}} \qquad \llbracket\, \Gamma \vdash a : A \,\rrbracket \triangleq \llbracket\, \Gamma \,\rrbracket \vdash \llbracket\, \Gamma \triangleright a \,\rrbracket : \llbracket\, \Gamma \triangleright A \,\rrbracket^{\mathrm{TY}}$$

We note that the translation of a judgment does not depend on its derivation in $\mathtt{Ob}^+$: as in [ACV96], we can thus avoid coherence issues in our proofs.

Theorem 1 (Validation of Typing). *If $\Gamma \vdash a$ is derivable in $\mathtt{Ob}^+$, then:*

1. $\llbracket\, \Gamma \,\rrbracket \vdash \llbracket\, \Gamma \triangleright a \,\rrbracket : \llbracket\, \Gamma \triangleright A \,\rrbracket^{\mathrm{TY}}$ *is derivable in* $\mathtt{F}_{\omega<:\mu}$.
2. *if* $a \xrightarrow{obj} b$, *then* $\llbracket\, \Gamma \triangleright a \,\rrbracket \xrightarrow{fun} \llbracket\, \Gamma \triangleright b \,\rrbracket$.

Theorem 2 (Computational Adequacy). *Let a be an $\mathtt{Ob}^+$ term such that $\emptyset \vdash a : A$ is derivable in $\mathtt{Ob}^+$. Then $a \Downarrow_{obj}$ if and only if $\llbracket\, \emptyset \triangleright a \,\rrbracket \Downarrow_{fun}$.*

6 Subtyping and Nonextensible Objects

The combination of object extension with subtyping has been studied from two orthogonal points of view in the literature: either limit subtyping in the presence of object extension, or distinguish extensible from nonextensible objects and disallow subtyping on the former while allowing it on the latter. Below, we focus on the second approach, deferring a discussion on the first to the full paper.

The idea of distinguishing between extensible and nonextensible objects was first proposed by Fisher and Mitchell in [FM95], to which the reader is referred to for details. Below, instead, we show that this idea allows different subtype relations to be formalized uniformly within the same framework.

Nonextensible objects are accounted for in $\mathtt{Ob}^{+}$ by introducing new types, contexts, and judgments as in the system $\mathtt{Ob}^{+}_{<:}$ (cf. Appendix A).

Table 4: Translation for $\mathtt{Ob}^{+}_{<:}$.

Types and Contexts	Judgments
$[\![\Gamma,' \mathtt{X} <: A, \Gamma'' \rhd \mathtt{X}]\!]^{\mathrm{TY}} \triangleq \mathtt{X}$	$[\![\Gamma \vdash A <: B]\!] \triangleq [\![\Gamma]\!] \vdash [\![\Gamma \rhd A]\!]^{\mathrm{TY}} <: [\![\Gamma \rhd B]\!]^{\mathrm{TY}}$
$[\![\Gamma, \mathtt{X} <: A]\!] \triangleq [\![\Gamma]\!], \mathtt{X} <: [\![\Gamma \rhd A]\!]^{\mathrm{TY}}$	

A further clause handles the translation of nonextensible object types: the format of this clause depends on how these types and the corresponding subtyping relation are defined. Below, we illustrate two cases.

Covariant Subtyping à la Fisher & Mitchell'95. The new types have the form $\mathtt{obj(X)}\langle\!\langle m_i{:}B_i\{\mathtt{X}\}\rangle\!\rangle^{i\in[1..n]}$, and their reading is similar to that of the `pro`-types of Section 2: unlike `pro`-typed objects, however, `obj`-typed objects may *not* be modified or extended from the outside. `pro` and `obj` types are ordered by subtyping, as established by the rule (Sub `probj FM95`) (in Appendix). Informally, `pro`-types may only be promoted to `obj`-types, not to other `pro`-types: hence only reflexive subtyping is available for `pro`-type, as required for the soundness of method addition and override. This subtyping rule allows subtyping both in *width* and *depth*: since elements of `obj`-types may not be overridden or extended, this powerful form of subtyping is sound. We note that the covariance condition $\Gamma, \mathtt{Y}, \mathtt{X} <: \mathtt{Y} \vdash B_i\{\mathtt{X}\} <: B'_i\{\mathtt{Y}\}$ is required also for the subtyping $\mathtt{pro(X)}\langle\!\langle m_i{:}B_i\{\mathtt{X}\}\rangle\!\rangle^{i\in[1..n]} <: \mathtt{obj(Y)}\langle\!\langle m_i{:}B_i\{\mathtt{Y}\}\rangle\!\rangle^{i\in[1..n]}$: as discussed in [FM95] covariance is crucial for subject-reduction: our translation, given below, explains why it is generally required for soundness.

Translation for `obj` types.

$$[\![\Gamma \rhd \mathtt{obj(X)}\langle m_i : B_i\{\mathtt{X}\}\rangle^{i\in[1..n]}]\!]^{\mathrm{TY}} \triangleq \mu(\mathtt{X})[m_i^{sel} : [\![\Gamma, \mathtt{X} \rhd B_i\{\mathtt{X}\}]\!]^{\mathrm{TY}}]^{i\in[1..n]}$$

The translation (which coincides with the standard recursive-record encoding) explains why `obj`-typed objects may not be extended or overridden: this is easily seen once we note that their type hides the polymorphic methods and the updaters. Self-inflicted updates, instead, are still allowed, as in [FM95]. This also explains why subtyping between `pro` and `obj` types is only allowed to covariant occurrences of the recursion variable. To exemplify, consider a term $e_1 : [\![\, \mathtt{pro}(x)\langle\!\langle m : \mathtt{X}{\to}B \rangle\!\rangle \,]\!]^{\mathrm{TY}}$, and assume that we allow e_1 to be viewed as an element of $[\![\, \mathtt{obj}(x)\langle\!\langle m : \mathtt{X}{\to}B \rangle\!\rangle \,]\!]^{\mathrm{TY}}$. Now, given $e_2 : [\![\, \mathtt{obj}(x)\langle\!\langle m : \mathtt{X}{\to}B \rangle\!\rangle \,]\!]^{\mathrm{TY}}$, the interpretation of $e_1 \Leftarrow m(e_2)$ is not sound, as the code of m in e_1 could use a *self-inflicted* update that is not available in the code for m in e_2 (consider that e_2 may not have the polymorphic methods available in e_1).

Theorem 3 (Validation of ***Fisher-Mitchell*** **Subtyping).** *If $\Gamma \vdash A <: B$ is derivable in $\mathtt{Ob}^{+}_{<:}$, (using (Sub* `probj FM95`*) for object subtyping) then the judgment $[\![\, \Gamma \,]\!] \vdash [\![\, \Gamma \triangleright A \,]\!]^{\mathrm{TY}} <: [\![\, \Gamma \triangleright B \,]\!]^{\mathrm{TY}}$ is derivable in $\mathtt{F}_{\omega<:\mu}$.*

Invariant Subtyping for Covariant Self Types à la Abadi & Cardelli'96. Covariant Self Types, denoted here by the type expression $\mathtt{obj}_{\mathtt{AC}}(\mathtt{X})\langle\!\langle m_i : B_i\{\mathtt{X}\} \rangle\!\rangle^{i\in[1..n]}$ are described in [AC96] (cf. Chaps. 15, 16). They share several features with the `obj`-types of [FM95], notably the fact that both describe collections of nonextensible objects. However, they have important specificities: (*i*) method override is a legal operation on elements of $\mathtt{obj}_{\mathtt{AC}}$ types, and (*ii*) subtyping over $\mathtt{obj}_{\mathtt{AC}}$ types is only allowed in *width*, and defined by the rule (Sub `probj AC96`) (cf. Appendix). A translation that validates that rule is given below:

Translation of $\mathtt{obj}_{\mathtt{AC}}$ Types

Let $A \equiv \mathtt{obj}_{\mathtt{AC}}(\mathtt{X})\langle m_i : B_i\{\mathtt{X}\}\rangle^{i\in[1..n]}$, and let $[\![\, \Gamma \triangleright A \,]\!]^{\mathrm{IN}}$ be defined as in Table 1.

$$\begin{aligned}[\![\, \Gamma \triangleright A \,]\!]^{\mathrm{TY}} \triangleq \mu(\mathtt{X})[\; & m_i^{upd} : (\forall(\mathtt{U} \le [\![\, \Gamma \triangleright A \,]\!]^{\mathrm{IN}})\mathtt{U}^* \to [\![\, \Gamma, \mathtt{X} \triangleright B_i\{\mathtt{X}\} \,]\!]^{\mathrm{TY}}\{\mathtt{X}{:=}\mathtt{U}^*\}) \to \mathtt{X}, \\ & m_i^{sel} : [\![\, \Gamma, \mathtt{X} \triangleright B_i\{\mathtt{X}\} \,]\!]^{\mathrm{TY}}]^{i\in[1..n]}\end{aligned}$$

Note how the updaters are exposed by the translation, thus making the translation of overrides well typed. Each of the component B_i is invariant in the translated type, as a result of a contravariant occurrence in the updater's type, and of a covariant occurrence in the selector's type.

Theorem 4 (Validation of ***Abadi-Cardelli*** **Subtyping).** *If $\Gamma \vdash A <: B$ is derivable in $\mathtt{Ob}^{+}_{<:}$, (using (Sub* `probj AC96`*) for object subtyping) then the judgment $[\![\, \Gamma \,]\!] \vdash [\![\, \Gamma \triangleright A \,]\!]^{\mathrm{TY}} <: [\![\, \Gamma \triangleright B \,]\!]^{\mathrm{TY}}$ is derivable in $\mathtt{F}_{\omega<:\mu}$.*

Invariant Subtyping. In [ACV96], an encoding is presented that validates invariant subtyping for object types, without requiring the covariance restriction for the component types. However, as discussed in [AC96], covariance is critical for sound method invocations: briefly, the problem arises with binary methods, since the use of bounded abstraction in the coding of the binder $\mathtt{obj}_{\mathtt{AC}}$ makes the type of self *unique*, hence different from any other type. The same problem affects the coding of [ACV96]: only covariant methods may be effectively invoked.

An interpretation with the same properties may be obtained from our translation. Given the type $\mathtt{obj}_{\mathtt{AC}}(\mathtt{X})\langle\!\langle m_i{:}B_i\{\mathtt{X}\}\rangle\!\rangle^{i\in[1..n]}$, invariant subtyping may be rendered by exposing the updaters of all the m_i's methods, while hiding the selectors of all the m_i's whose type B_i is not covariant in the bound variable. This translation would be the exact equivalent of that proposed in [ACV96]: it would validate invariant subtyping, and allow invocation only for covariant methods.

7 Related Work

The idea to split methods into different components is inspired by the object encoding of [ACV96]. That translation applies only to nonextensible objects, which are encoded by a combined use of recursive and bounded existential types, subsequently named `ORBE` encoding [BCP97]. Our translation, instead, uses a combination of recursion and universal quantification to render `MyType` polymorphism. We are then able to obtain a corresponding translation for nonextensible objects with essentially equivalent results as [ACV96].

A variant of the `ORBE` encoding that does not use existential types is proposed in [AC96] (Chap. 18): our translation can be viewed as an extension of that encoding to handle primitives of method addition.

Other, more recent papers have studied object encodings. In [Cra98], Crary proposed a simpler alternative to the `ORBE` encoding for nonextensible objects based on a combination of existential and intersection types. In [Vis98] Visvanathan gives a full-abstract translation for first-order objects with recursive types (but no Self Types). Again, the translation does not handle extensible objects. In [BDZ99], Boudol and Dal-Zilio study an encoding for extensible objects that relies on essentially the same idea used in our interpretation, namely the representation of extensible objects as a pair of a generator and a non extensible object. The difference is that [BDZ99] uses extensible records in the target calculus to model object generators in ways similar to [Coo89].

References

[AC95] M. Abadi and L. Cardelli. On Subtyping and Matching. In *Proceedings of ECOOP'95: European Conference on Object-Oriented Programming*, volume 952 of *LNCS*, pages 145–167. Springer–Verlag, August 1995.

[AC96] M. Abadi and L. Cardelli. *A Theory of Objects.* Monographs in Computer Science. Springer, 1996.

[ACV96] M. Abadi, L. Cardelli, and R. Viswanathan. An Iterpretation of Objects and Object Types. In *Proc. of POPL'96*, pages 396–409, 1996.

[BB98] V. Bono and M. Bugliesi. Matching for the Lambda Calculus of Objects. *Theoretical Computer Science*, 1998. To appear.

[BCP97] K. Bruce, L. Cardelli, and B. Pierce. Comparing Object Encodings. In *Proc. of TACS'97*, volume 1281 of *Lecture Notes in Computer Science*, pages 415–438. Springer-Verlag, 1997.

[BDZ99] G. Boudol and S. Dal-Zilio. An interpretation of extensible objects. In *Proceedings of FCT'99*, 1999.

[BL95] V. Bono and L. Liquori. A Subtyping for the Fisher-Honsell-Mitchell Lambda Calculus of Objects. In *Proc. of CSL*, volume 933 of *Lecture Notes in Computer Science*, pages 16–30. Springer-Verlag, 1995.

[Bru94] K.B. Bruce. A Paradigmatic Object-Oriented Programming Language: Design, Static Typing and Semantcs. *Journal of Functional Programming*, 1(4):127–206, 1994.

[Coo87] W. Cook. A Self-ish Model of Inheritance. Manuscript, 1987.

[Coo89] W.R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.

[Cra98] K. Crary. Simple, efficient object encoding using intersection types. Technical report, Cornell University, April 1998.

[FHM94] K. Fisher, F. Honsell, and J. C. Mitchell. A Lambda Calculus of Objects and Method Specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994.

[FM95] K. Fisher and J. C. Mitchell. A Delegation-based Object Calculus with Subtyping. In *Proc. of FCT*, volume 965 of *Lecture Notes in Computer Science*, pages 42–61. Springer-Verlag, 1995.

[Kam88] S. Kamin. Inheritance in Smalltalk-80: a denotational definition. In *Proc. of POPL'88*, pages 80–87. ACM Press, 1988.

[Liq97] L. Liquori. An Extended Theory of Primitive Objects: First Oder System. In *Proc. of ECOOP*, volume 1241 of *Lecture Notes in Computer Science*, pages 146–169. Springer-Verlag, 1997.

[PT94] B. Pierce and D. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–248, 1994.

[Rem97] D. Remy. From classes to objects, via subsumption. Technical report, INRIA, 1997. Also in Proceeding of ESOP'98.

[Vis98] R. Viswanathan. Full abstraction for first-order objects with recursive types and subtyping. In *Proc. of LICS'98*, pages 380–391, 1998.

A The Source Calculus

Reduction

$$a \equiv \varsigma(\mathtt{U}, A)\langle m_i = \varsigma(x : \mathtt{U})b_i\rangle^{i\in[1..n]}$$

(Call) $(j \in [1..n])$

$a \Leftarrow m_j \ \succ\ [a/x]b_j$

(Extend) $m_{n+1} \notin \{m_1, .., m_n\}$

$a \longleftarrow\!\!+\ m_{n+1} = \varsigma(\mathtt{U}, A')\varsigma(x : \mathtt{U})b \ \succ\ \varsigma(\mathtt{U}, A')\langle m_i = \varsigma(x : \mathtt{U})b_i\rangle^{i\in[1..n+1]}$

(Override) $(j \in [1..n])$

$a \leftarrow m_j = \varsigma(\mathtt{U}, A)\varsigma(x : \mathtt{U})b \ \succ\ \varsigma(\mathtt{U}, A)\langle m_i = \varsigma(x : \mathtt{U})b_i, m_j = \varsigma(x : \mathtt{U})b\rangle^{i\in[1..n]-\{j\}}$

Results $\quad v ::= \varsigma(\mathtt{U}, A)\langle m_i = \varsigma(x : \mathtt{U})b_i\rangle^{i\in[1..n]}$

Context Formation – $\mathtt{Ob}^+$

$$\text{(Ctx } \oslash\text{)}\ \frac{}{\emptyset \vdash *} \qquad \text{(Ctx } x\text{)}\ \frac{\Gamma \vdash A \quad x \notin Dom(\Gamma)}{\Gamma, x : A \vdash *} \qquad \text{(Ctx Match)}\ \frac{\Gamma \vdash A \quad \mathtt{U} \notin Dom(\Gamma)}{\Gamma, \mathtt{U} \lessdot\!\# A \vdash *} \qquad \text{(Ctx } \mathtt{X}\text{)}\ \frac{\Gamma \vdash * \quad \mathtt{X} \notin Dom(\Gamma)}{\Gamma, \mathtt{X} \vdash *}$$

Type formation – $\mathtt{Ob}^{+}$

(Type Match $\mathtt{U}$)
$$\frac{\Gamma', \mathtt{U} \mathrel{<\!\#} A, \Gamma'' \vdash *}{\Gamma', \mathtt{U} \mathrel{<\!\#} A, \Gamma'' \vdash \mathtt{U}}$$

(Type $\mathtt{X}$)
$$\frac{\Gamma', \mathtt{X}, \Gamma'' \vdash *}{\Gamma', \mathtt{X}, \Gamma'' \vdash \mathtt{X}}$$

(Type $\mathtt{pro}$)
$$\frac{\Gamma, \mathtt{X} \vdash A_i}{\Gamma \vdash \mathtt{pro}(\mathtt{X})\langle m_i{:}A_i\rangle^{i \in I}}$$

Term Formation – $\mathtt{Ob}^{+}$

The notation $\Gamma\langle A\rangle$ in (Val Override) is defined as follows:

$$\begin{array}{lll} \Gamma\langle A\rangle & \equiv A & \text{if } A \text{ is a } \mathtt{pro}\text{-type;} \\ \Gamma\langle A\rangle & \equiv A' & \text{if } A \equiv \mathtt{U}' \text{ and } \mathtt{U}' \mathrel{<\!\#} A' \in \Gamma. \end{array}$$

(Val x)
$$\frac{\Gamma', x : A, \Gamma'' \vdash *}{\Gamma', x : A, \Gamma'' \vdash x : A}$$

(Val Override)
$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash A \mathrel{<\!\#} \mathtt{pro}(\mathtt{X})\langle m : B\{\mathtt{X}\}\rangle \\ \Gamma, \mathtt{U} \mathrel{<\!\#} \Gamma\langle A\rangle, x : \mathtt{U} \vdash b : B\{\mathtt{U}\}}{\Gamma \vdash a \leftarrow m = \varsigma(\mathtt{U}, A)\varsigma(x : \mathtt{U})b : A}$$

(Val Send)
$$\frac{\Gamma \vdash e : A \\ \Gamma \vdash A \mathrel{<\!\#} \mathtt{pro}(\mathtt{X})\langle m{:}B\{\mathtt{X}\}\rangle}{\Gamma \vdash e \Leftarrow m : B\{A\}}$$

(Val Extend)
$$\frac{(A \equiv \mathtt{pro}(\mathtt{X})\langle m_i{:}B_i\{\mathtt{X}\}\rangle^{i \in [1..n]} \\ A^{+} \equiv \mathtt{pro}(\mathtt{X})\langle m_i{:}B_i\{\mathtt{X}\}\rangle^{i \in [1..n+1]}) \\ \Gamma \vdash a : A \quad \Gamma, \mathtt{U} \mathrel{<\!\#} A^{ext}, x : \mathtt{U} \vdash b : B_{n+1}\{\mathtt{U}\}}{\Gamma \vdash a \leftarrow\!\!+\ \varsigma(\mathtt{U}, A^{+})m_{n+1} = \varsigma(x : \mathtt{U})b : A^{+}}$$

(Val Object)
$$\frac{(A \equiv \mathtt{pro}(\mathtt{X})\langle m_i{:}B_i\{\mathtt{X}\}\rangle^{i \in [1..n]}) \\ \Gamma, \mathtt{U} \mathrel{<\!\#} A, x : \mathtt{U} \vdash b_i : B_i\{\mathtt{U}\} \qquad \forall i \in [1..n]}{\Gamma \vdash \varsigma(\mathtt{U}, A)\langle m_i = \varsigma(x : \mathtt{U})b_i\rangle^{i \in [1..n]} : A}$$

Matching – $\mathtt{Ob}^{+}$

(Match $\mathtt{U}$)
$$\frac{\Gamma', \mathtt{U} \mathrel{<\!\#} A, \Gamma'' \vdash *}{\Gamma', \mathtt{U} \mathrel{<\!\#} A, \Gamma'' \vdash \mathtt{U} \mathrel{<\!\#} A}$$

(Match Refl)
$$\frac{\Gamma', \mathtt{U} \mathrel{<\!\#} A, \Gamma'' \vdash \mathtt{U}}{\Gamma', \mathtt{U} \mathrel{<\!\#} A, \Gamma'' \vdash \mathtt{U} \mathrel{<\!\#} \mathtt{U}}$$

(Match Trans)
$$\frac{\Gamma \vdash u \mathrel{<\!\#} B \quad \Gamma \vdash B \mathrel{<\!\#} A}{\Gamma \vdash \mathtt{U} \mathrel{<\!\#} A}$$

(Match $\mathtt{pro}$)
$$\frac{\Gamma \vdash \mathtt{pro}(\mathtt{X})\langle m_i{:}B_i\{\mathtt{X}\}\rangle^{i \in [1..n+k]}}{\Gamma \vdash \mathtt{pro}(\mathtt{X})\langle m_i{:}B_i\{\mathtt{X}\}\rangle^{i \in [1..n+k]} \mathrel{<\!\#} \mathtt{pro}(\mathtt{X})\langle m_i{:}B_i\{\mathtt{X}\}\rangle^{i \in [1..n]}}$$

Context and Type Formation – $\mathtt{Ob}^{+}_{<:}$

(Ctx Sub)
$$\frac{\Gamma \vdash A \quad \mathtt{U} \notin Dom(\Gamma)}{\Gamma, \mathtt{U} <: A \vdash *}$$

(Type $\mathtt{obj}$)
$$\frac{\Gamma, \mathtt{X} \vdash B_i \quad \forall i \in [1..n]}{\Gamma \vdash \mathtt{obj}(\mathtt{X})\langle m_i{:}B_i\rangle^{i \in [1..n]}}$$

Term Formation – $\mathtt{Ob}^{+}_{<:}$

(Val Send $\mathtt{Obj}$)
$$\frac{\Gamma \vdash e : A \quad (A \equiv \mathtt{obj}(\mathtt{X})\langle m_i{:}B_i\{\mathtt{X}\}\rangle^{i \in [1..n]}, \ \ j \in [1..n])}{\Gamma \vdash e \Leftarrow m_j : B_j\{A\}}$$

(Val Subsumption)
$$\frac{\Gamma \vdash e : A \quad \Gamma \vdash A <: B}{\Gamma \vdash e : B}$$

Subtyping – $\mathtt{Ob}^{+}_{<:}$

(Sub $\mathtt{U}$)

$$\frac{\Gamma', \mathtt{U} <: A, \Gamma'' \vdash *}{\Gamma', \mathtt{U} <: A, \Gamma'' \vdash \mathtt{U} <: A}$$

(Sub Refl)

$$\frac{\Gamma \vdash A}{\Gamma \vdash A <: A}$$

(Sub Trans)

$$\frac{\Gamma \vdash A_1 <: A_2 \quad \Gamma \vdash A_2 <: A_3}{\Gamma \vdash A_1 <: A_3}$$

(Sub $\mathtt{probj\ FM95}$)

$$\frac{\Gamma, \mathtt{Y}, \mathtt{X} <: \mathtt{Y} \vdash B'_i\{\mathtt{X}\} <: B_i\{\mathtt{Y}\} \qquad \forall i \in [1..n]}{\Gamma \vdash \mathtt{probj}(\mathtt{X})\langle m_i{:}B'_i\{\mathtt{X}\}\rangle^{i\in[1..n+k]} <: \mathtt{obj}(\mathtt{Y})\langle B_i\{\mathtt{Y}\}\rangle^{i\in[1..n]}}$$

(Sub $\mathtt{probj\ AC96}$)

$$\frac{\Gamma, \mathtt{X} \vdash B_i\{\mathtt{X}\} \quad B_i \ \mathit{covariant\ in}\ \mathtt{X} \quad \forall i \in [1..n+k]}{\Gamma \vdash \mathtt{probj}_{\mathtt{AC}}(\mathtt{X})\langle m_i : B_i\{\mathtt{X}\}\rangle^{i\in[1..n+k]} <: \mathtt{obj}_{\mathtt{AC}}(\mathtt{X})\langle m_i : B_i\{\mathtt{X}\}\rangle^{i\in[1..n]}}$$

B The Target Calculus $\mathtt{F}_{\omega<:\mu}$

Syntax

$\mathcal{K}$ ::=	*Kinds*	
	$\mathcal{T}$	types
	$\mathcal{K} \Rightarrow \mathcal{K}$	type operators
$\mathbb{A}, \mathbb{B}$ ::=	*Constructors*	
	$\mathtt{X}$	constructor variable
	$\top$	greatest constructor of kind $\mathcal{T}$
	$\mathbb{A} \to \mathbb{B}$	function type
	$[m_1 : \mathbb{B}, \ldots, m_k : \mathbb{B}]$	record type
	$\forall(\mathtt{X} <: \mathbb{A} :: \mathcal{K})\mathbb{A}$	bounded universal type
	$\mu(\mathtt{X})\mathbb{A}$	recursive type
	$\lambda(\mathtt{X} :: \mathcal{K})\mathbb{B}$	operator
	$\mathbb{B}(\mathbb{A})$	operator application
e ::=	*Expressions*	
	x	variables
	$\lambda(x : \mathbb{A})\, e$	abstraction
	$e_1\, e_2$	application
	$\Lambda(\mathtt{X} <: \mathbb{A} :: \mathcal{K})\, e$	type-abstraction
	$e\, \mathbb{A}$	type-application
	$[m_1 = e_1, \ldots, m_k = e_k]$	record
	$e.m$	record selection
	$\mathtt{fold}(\mathbb{A}, e)$	recursive fold
	$\mathtt{unfold}(e)$	recursive unfold
	$\mathtt{let}\ x = e_1\ \mathtt{in}\ e_2$	local definition
	$\mathtt{letrec}\ f(x : \mathbb{A}) : \mathbb{B} = e_1\ \mathtt{in}\ e_2$	recursive local definition

Reduction

(β_1)	$(\lambda(x : \mathbb{A})e_1)\, e_2$	$\leadsto$ $[e_2/x]\, e_1$	(*select*)	$[m_i = e_i]^{i\in[1..n]}.m_j \leadsto e_j$	$(j \in [1..n])$
(β_2)	$(\Lambda(\mathtt{X} \leq \mathbb{A})e_1)\, \mathbb{B}$	$\leadsto$ $[\mathbb{B}/\mathtt{X}]\, e_1$	(*unfold*)	$\mathtt{unfold}(\mathtt{fold}(\mathbb{A}, e)) \leadsto e$	

Results $\quad v ::= \lambda(x : \mathbb{A})\, e \mid [m_1 = e_1, \ldots, m_k = e_k] \mid \mathtt{fold}(\mathbb{A}, e) \mid \Lambda(\mathtt{X} <: \mathbb{A} :: \mathcal{K})\, e$