An Interpretation of Objects and Object Types

Martín Abadi*
Digital Systems Research Center
ma@pa.dec.com

Luca Cardelli*
Digital Systems Research Center
luca@pa.dec.com

Ramesh Viswanathan†
Isaac Newton Institute for Mathematical Sciences
R.Viswanathan@newton.cam.ac.uk

Abstract

We present an interpretation of typed object-oriented concepts in terms of well-understood, purely procedural concepts. More precisely, we give a compositional subtype-preserving translation of a basic object calculus supporting method invocation, functional method update, and subtyping, into the polymorphic λ-calculus with recursive types and subtyping. The translation techniques apply also to an imperative version of the object calculus which includes in-place method update and object cloning. Finally, the translation easily extends to "Self types" and other interesting object-oriented constructs.

1 Introduction

Object-oriented programming languages have introduced numerous ideas, structures, and techniques. Although these contributions are not always conceptually clear (or even sound), they are often original and useful. One of the most basic contributions is the notion of *self*; the operations associated with an object (its *methods*) can refer to the object as self, and invoke other operations by indirecting through self, with dynamic dispatch. A related contribution is the notion of *subsumption*: an object can be replaced (subsumed by) any object that supports the same or more operations; in typed languages, subsumption is systematized in rules for *subclassing* and *subtyping*.

Object-oriented programming is not limited to object-oriented languages. One can emulate objects in some procedural languages, such as Scheme and C. So it is possible that, despite its originality, object-oriented programming can be reduced to procedural programming. Such a reduction is not straightforward. Interesting difficulties arise at the level of types: the most natural definition of objects as records of functions (the *self-application semantics* [Kam88]) does not validate the expected subtypings, so subsumption is blocked.

In this paper we develop an interpretation of object-oriented programming in terms of procedural programming (specifically, in terms of a fairly standard λ-calculus with subtyping). We show a translation of objects into records of functions, and a translation of object types into types built up from record types, existential types, and recursive types. These ingredients should not be surprising, but their combination is new. The translation is faithful in that it respects operational semantics, typing rules, and subtyping rules; it yields a syntactic proof of soundness for those rules.

In order to make this interpretation both manageable and precise, we develop it in the context of object calculi [AC94b]. Object calculi are formalisms analogous to λ-calculi, based on objects rather than on functions. Their only primitives are objects, method invocation, method update, and (for imperative calculi) cloning. Method update is the most unusual of these, but forms of method update do appear in several languages [Lie81, And92, Tai92, MGV92, MMPN93, ALBC+93, App93, Car95]. The primitives are quite expressive: they allow representations of both class-based and object-based notions, for example classes, subclasses, protection, prototyping, and mode switching. When typed, object calculi include a subtyping relation and a subsumption rule.

We believe that our interpretation of objects is compelling for several reasons.

- First, it makes precise the vague intuition that objects have something to do with abstract data types and recursive types. That intuition has been important in previous works that studied λ-calculi with subtyping and used them to emulate objects to various extents (see the next section). The target calculus of our translation is the result of those previous works.
- The translation is sufficiently complicated to confirm that objects provide a useful abstraction independent of procedural concepts. On the other hand, it is simple enough to serve as an explanation of objects in terms of well-understood constructs.
- Finally, the translation is not limited to one particular object calculus. With some modifications, it applies both to functional and imperative execution models; and it can be adapted to account for "Self types" and "structural rules" [AC95b], which are operationally sound but unsound in common denotational models.

In the next section we review some of the background, describing related works, and give an informal overview of

*Address: Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, California 94301, U.S.A.

†Supported by NSF Grant CCR-9303099 and a Rosenbaum Fellowship. Address: Isaac Newton Institute for Mathematical Sciences, 20 Clarkson Road, Cambridge, CB3 0EH, U.K.

To appear in the Proceedings of the 23rd Annual ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages, January 1996, St. Petersburg Beach, Florida.

our interpretation. Section 3 defines a first version of the translation precisely; Section 4 gives an imperative translation; Section 5 deals with additional type constructs.

2 Informal Review and Overview

We first review a basic untyped object calculus [AC94b] since we use its notation in what follows. In this calculus, an object $[l_1 = \varsigma(x_1)b_1, \ldots, l_n = \varsigma(x_n)b_n]$ is a collection of methods $\varsigma(x_1)b_1, \ldots, \varsigma(x_n)b_n$ with respective names $l_1, \ldots, l_n$; the method bodies are $b_1, \ldots, b_n$, and the variables $x_1, \ldots, x_n$ denote self. The order of the methods does not matter. The only operations on objects are method invocation and method update. If o is the object $[l_1 = \varsigma(x_1)b_1, \ldots, l_n = \varsigma(x_n)b_n]$, then the invocation of its method l_i, written $o.l_i$, consists in replacing o for x_i in b_i; the method update $o.l_i \Leftarrow \varsigma(x)b$ yields an object like o but where we have $l_i = \varsigma(x)b$. For now, we take the update construct to be functional, that is, to create a new object. Fields (instance variables) are easily expressible as methods that do not use their self parameters.

This untyped calculus is the basis for a first-order calculus with subtyping, called $\mathbf{Ob_{1<:}}$. In $\mathbf{Ob_{1<:}}$, an object type $[l_1 : B_1, \ldots, l_n : B_n]$ is the type of objects which have methods $l_1, \ldots, l_n$ that, when invoked, result in values of types $B_1, \ldots, B_n$ respectively.

A characteristic of object-oriented languages is that an object with more methods can be used wherever an object with fewer methods is expected. In $\mathbf{Ob_{1<:}}$, this is supported through a subtyping relation $<:$. An object type with more methods is a subtype of an object type with fewer methods provided that the common methods have exactly the same result types; for example, $[l_1 : B_1, l_2 : B_2] <: [l_1, B_1]$ for any types B_1 and B_2.

The self-application semantics [Kam88] provides a satisfactory explanation of untyped objects as records of functions. Let us write $\{l_1 = a_1, \ldots, l_n = a_n\}$ for the record with fields $l_1, \ldots, l_n$ with $a_1, \ldots, a_n$ as values; $a \cdot l$ for extracting the field l of record a; and $a \cdot l := b$ for updating the field l of record a to be b. In the self-application semantics, a method is a function of its self parameter; an object is a record of such functions; method invocation is field selection plus self-application; method update is record update:

$$\begin{array}{rcl} [l_1 = \varsigma(x_i)b_i \ ^{i\in 1\ldots n}] & \triangleq & \{l_i = \lambda(x_i)b_i \ ^{i\in 1\ldots n}\} \\ o.l_j & \triangleq & o \cdot l_j(o) \\ o.l_j \Leftarrow \varsigma(y)b & \triangleq & o \cdot l_j := \lambda(y)b \end{array}$$

This interpretation respects the operational behavior of both method invocation and method update.

Unfortunately, the self-application semantics does not extend to typed systems such as $\mathbf{Ob_{1<:}}$. In particular, it does not validate the essential subtypings between object types. Following the self-application semantics, one would naturally interpret the object type $A \equiv [l_i : B_i \ ^{i\in 1\ldots n}]$ as a recursive record type, the solution to the type equation:

$$A \ = \ \{l_i : A \to B_i \ ^{i\in 1\ldots n}\}$$

where $\{l_i : C_i \ ^{i\in 1\ldots n}\}$ denotes the evident record type. Because of the contravariant occurrences of the object type A, we do not obtain subtypings valid in $\mathbf{Ob_{1<:}}$ such as $[l_1 : B_1, l_2 : B_2] <: [l_1 : B_1]$.

In part because of this difficulty, there have been several other interpretations of objects. Most of them were defined as ways of emulating objects in procedural settings, rather than as precise translations, so it is somewhat hard to give a full account of their scope. In short, many of them contributed interesting and useful techniques but they all suffer from limitations. The recursive-record semantics [Car88] validates the expected subtypings, but it does not model method update (or even field update); the generator semantics [Coo89] deals with update, at the cost of separating objects from object generators. The existential interpretation of [PT94, HP95] also validates the expected subtypings, and models class-based constructs where methods and fields are rigidly separated and it is only the fields that can be updated; unfortunately, the translation of objects is type-directed, and rather elaborate. An imperative interpretation [ESTZ95] can solve the problems of the self-application semantics with judicious side-effects; its main limitation is that it does not model the cloning construct of object-based imperative languages. Finally, some interpretations give up on subtyping altogether, and reduce it to coercion functions [AC94a, Rém94]; the coercion functions are cumbersome, destroy the flavor of the original programs, and preclude an explanation of object subtyping in terms of more primitive subtyping relations.

At this point, a possible conclusion is that it is easy to understand the computational behavior of objects, but that their desired typing and subtyping properties make them fundamentally different from records and functions. It was this view that originally led to the formulation of object calculi. Our interpretation of objects, which we discuss next, sheds some new light on this matter. It does provide a rather complete account of objects in terms of records and functions. It applies both to class-based and object-based constructs, places no restrictions on method update, and validates the expected subtypings. On the other hand, because it is not straightforward, it does not remove the usefulness of object calculi as a setting for studying object-oriented concepts.

In this paper we develop translations of several object calculi into λ-calculi. The translations are faithful in that they respect the operational semantics, typing rules, and subtyping rules of the object calculi. The first translation maps $\mathbf{Ob_{1<:}}$ into $\mathbf{F}_{<:\mu}$, the polymorphic λ-calculus with subtyping and recursive types; we preview this translation next, explaining informally how it treats types.

Consider a type $A \equiv [l_1 : B_1, \ldots, l_n : B_n]$. Because of subsumption, an object o containing additional methods besides $l_1, \ldots, l_n$ can be an element of A. If we think of the "true type" of o as the type listing all its methods, then the type A only partially reveals the true type of o; what is publicly visible are only the methods $l_1, \ldots, l_n$. We therefore take the translation A^* of an object type A to be a type abstraction with representation type the true type of the object. Using the notation $\exists(X<:C)B$ for a type abstraction with an interface B and an unknown representation type X that is a subtype of C, we define A^* as a recursive type, with the following equation:

$$\begin{array}{rcl} A^* & = & \exists(X<:A^*)\{l_i^{sel} : (X \to B_i^*) \ ^{i\in 1\ldots n}, \\ & & \quad l_i^{upd} : (X \to B_i^*) \to X \ ^{i\in 1\ldots n}, \\ & & \quad self : X\} \end{array} \qquad (1)$$

The subtyping assumption $X<:A^*$ for the representation type expresses that the true type is known to be a subtype

Table 1: Operational Semantics of $\mathbf{Ob_{1<:}}$ and $\mathbf{F}_{<:\mu}$

$\mathbf{Ob_{1<:}}$	If $a \equiv [l_i = \varsigma(x_i : A)b_i\{x_i\}^{\ i\in 1\ldots n}]$, $j \in 1\ldots n$ (Eval Select) $a.l_j \rightsquigarrow b_j\{a\}$ (Eval Update) $a.l_j \Leftarrow \varsigma(x : A')b \rightsquigarrow [l_j = \varsigma(x : A)b, l_i = \varsigma(x_i : A)b_i^{\ i\in\{1\ldots n\}-\{j\}}]$
$\mathbf{F}_{<:\mu}$	(Eval Beta) $(\lambda(x : A)b\{x\})(a) \rightsquigarrow b\{a\}$ (Eval Beta2) $(\lambda(X<:A)b\{X\})(A') \rightsquigarrow b\{A'\}$ (Eval Record Select) $\{l_i = b_i^{\ i\in 1\ldots n}\} \cdot l_j \rightsquigarrow b_j$ for $j \in 1\ldots n$ (Eval Unfold) $\texttt{unfold}(\texttt{fold}(A, a)) \rightsquigarrow a$ (Eval Unpack) $\texttt{open}\ c\ \texttt{as}\ X<:A, x : B\{X\}\ \texttt{in}\ d\{X, x\} : D \rightsquigarrow d\{C, b\{C\}\}$ where $c \equiv \texttt{pack}\ X<:A' = C\ \texttt{with}\ b\{X\} : B'\{X\}$

of the object type. The field l_i^{sel} is the method l_i treated as a function of self. The field l_i^{upd} provides the ability to update method l_i—given a new method that is a function of self, it returns a new object. The field *self* is the object itself with all its methods (including the private ones); through *self*, the methods $l_1, \ldots, l_n$ can access methods not listed in the interface.

Each of the ingredients in this translation is necessary. In particular, the use of existential types in addition to recursive types is essential for getting the desired subtypings. Similarly, it is essential to model method update via a field l_i^{upd}: if method update were modeled by an update of the field l_i^{sel} then this would leave the field *self* unaffected, so the operational semantics would be distorted.

In the next section we detail this translation of $\mathbf{Ob_{1<:}}$. In Section 4, we consider an imperative version of the translation, which deals correctly with cloning but is in some ways simpler than the functional one thanks to side-effects. In Section 5, we show that the translation of $\mathbf{Ob_{1<:}}$ can be extended to account for richer object-type constructs with Self types and variance annotations.

3 Interpretation of an Object Calculus with Functional Update

In this section, we describe the translation of the object calculus $\mathbf{Ob_{1<:}}$ into the functional calculus $\mathbf{F}_{<:\mu}$. In Sections 3.1 and 3.2, we briefly describe the calculi $\mathbf{Ob_{1<:}}$ and $\mathbf{F}_{<:\mu}$, and in Section 3.3, we detail the translation of $\mathbf{Ob_{1<:}}$ into $\mathbf{F}_{<:\mu}$. The precise typing rules for the two calculi are given in Appendices A and B.

3.1 An Object Calculus: $\mathbf{Ob_{1<:}}$

The types of $\mathbf{Ob_{1<:}}$ are generated by the grammar:

$$A, B \ ::= \ \mathit{Top} \mid [l_i : B_i^{\ i\in 1\ldots n}]$$

where $n \geq 0$. The type *Top* is the supertype of all types and $[l_i : B_i^{\ i\in 1\ldots n}]$ is the type of objects with methods l_i returning results of type B_i. The terms of the calculus are similar to those of the untyped calculus described in Section 2 except that ς-bound variables have type annotations:

$$a, b \ ::= \ x \mid [l_i = \varsigma(x_i : A)b_i^{\ i\in 1\ldots n}] \mid a.l \mid a.l \Leftarrow \varsigma(x : A)b$$

A subset of the terms generated by this grammar are identified as well-typed terms by a set of typing rules described in Appendix A. The rules are used to derive judgements of the form $E \vdash \mathcal{J}$, where $\mathcal{J}$ is an assertion and E is an environment describing assumptions about the free variables in $\mathcal{J}$. The assertion $\diamond$ means that E is a well-formed environment, A means that A is a well-formed type, $A<:B$ means that A is a subtype of B, and $a : A$ means that a is a well-formed term of type A. An important rule, (Sub Object), states that A is a subtype of A' if A has all the method names given in A' and moreover the result types of these methods are exactly the same in A and A' (so object types are invariant in their component types).

The operational semantics is defined via a reduction system; it is free of side-effects. The primitive redexes given in Table 1 correspond to method invocation and method update; we write $b\{x\}$ to distinguish a variable x that may occur free in b, and $b\{a\}$ for the result of replacing x with a in b once x is clear from context. The one-step reduction relation $\longrightarrow_o$ is the congruence closure of $\rightsquigarrow$ (*i.e.*, we can reduce any subterm that is a redex); the many-step reduction relation $\longrightarrow\!\!\!\!\!\rightarrow_o$ is the reflexive, transitive closure of $\longrightarrow_o$. We define results to be terms of the form $[l_i = \varsigma(x_i : A)b_i^{\ i\in 1\ldots n}]$; we say that a closed term a converges, and write $a \Downarrow_o$, if there exists a result v such that $a \longrightarrow\!\!\!\!\!\rightarrow_o v$.

3.2 A Functional Calculus: $\mathbf{F}_{<:\mu}$

The system $\mathbf{F}_{<:\mu}$ is the standard extension of System $\mathbf{F}$ with recursive types and subtyping. While records and existentially quantified types are encodable in terms of the other constructs of $\mathbf{F}_{<:\mu}$, we present them as primitive for simplicity. System $\mathbf{F}_{<:\mu}$ is defined in detail in Appendix B; in this section, we describe informally some of its constructs.

Records are collections of fields with associated values; the only operation on records is field extraction (written $r \cdot l$). The basic types are function types and record types. A record type $\{l_i : B_i^{\ i\in 1\ldots n}\}$ lists the field names and the types of the values associated with them. Record types are covariant in their component types. We use the recursive type $\mu(X)B\{X\}$ to denote a solution to the type equation $X = B\{X\}$ where X could occur free in B. The isomorphism between $\mu(X)B\{X\}$ and its unfolding $B\{\mu(X)B\{X\}\}$ is given by the constructs `fold` and `unfold`: if a is of type $\mu(X)B\{X\}$ then $\texttt{unfold}(a)$ is of the unfolded type, and if b

is of the unfolded type then $\texttt{fold}(\mu(X)B\{X\}, a)$ is of type $\mu(X)B\{X\}$.

The existentially quantified type $\exists(X{<:}A)B\{X\}$ is the type of a term a (roughly) if there exists a type C that is a subtype of A for which a is a term of type $B\{C\}$. More formally, given a term a of type $B\{C\}$, the term $\texttt{pack}\ X <: A = C\ \texttt{with}\ a : B\{X\}$ has type $\exists(X{<:}A)B\{X\}$. What we achieve by packing a in a term of type $\exists(X{<:}A)B\{X\}$ is the hiding of information about the type C at which a realizes $\exists(X{<:}A)B\{X\}$. (Recall that data abstractions have existential types [MP88].) Given a term c of type $\exists(X{<:}A)B\{X\}$, we can access its "inside" by writing the term $\texttt{open}\ c\ \texttt{as}\ X{<:}A, y : B\{X\}\ \texttt{in}\ d : D$, where X stands for the representation type and y for the "inside". We can use X and y in d but the typing rules ensure that d cannot assume any information about X other than that it is a subtype of A, and the type D specified for d must not depend on the representation type X, *i.e.*, X cannot occur free in D.

Table 1 specifies reduction for $\mathbf{F}_{<:\mu}$. The one-step reduction relation $\longrightarrow_f$ is the congruence closure of $\rightsquigarrow$, and $\longrightarrow\!\!\!\!\!\rightarrow_f$ is the reflexive, transitive closure of $\longrightarrow_f$. As for $\mathbf{Ob_{1<:}}$, we distinguish certain terms as results; the set of results, defined in Appendix B, includes λ-abstractions and records. We say that a closed term a converges, and write $a \Downarrow_f$, if there exists a result v such that $a \longrightarrow\!\!\!\!\!\rightarrow_f v$.

Using recursive types, we can easily define a (call-by-name) fixed point operator. It is also routine to define `letrec`; we write $\texttt{letrec}\ f(x_1 : A_1)\cdots(x_n : A_n) : B = b\ \texttt{in}\ c$ to denote a recursive definition of a function f of type $A_1 \to \cdots \to A_n \to B$, used in the term c.

3.3 Translation

We are now ready to describe the translation of $\mathbf{Ob_{1<:}}$ into $\mathbf{F}_{<:\mu}$. The translation is in two parts. The first part is a translation of types which maps every type A of $\mathbf{Ob_{1<:}}$ to a type A^* of $\mathbf{F}_{<:\mu}$ and is defined by induction on the structure of types in $\mathbf{Ob_{1<:}}$.

$$\begin{array}{rcl} \mathit{Top}^* & = & \mathit{Top} \\ [l_i : B_i\ ^{i\in 1\ldots n}]^* & = & \mu(Y)\exists(X{<:}Y) \\ & & \quad \{l_i^{sel} : (X \to B_i^*)\ ^{i\in 1\ldots n}, \\ & & \quad\ l_i^{upd} : (X \to B_i^*) \to X\ ^{i\in 1\ldots n}, \\ & & \quad\ \mathit{self} : X\} \end{array}$$

The interpretation of object types given here is the same as that of Section 2, but here we use a μ rather than an equation (Equation (1)) for defining the existential type recursively. Note that in the translation of the object type $A \equiv [l_i : B_i\ ^{i\in 1\ldots n}]$, the field l_i^{sel} makes the type A^* covariant in B_i^* and the field l_i^{upd} makes it contravariant in B_i^*. Our interpretation thus explains the invariance of object types in their component types as arising from a covariance due to invocation and a contravariance due to update.

The key consequence of our translation of object types is that it gives the expected subtypings. More formally, we use the translation of types to define a mapping E^* for environments, and establish Theorem 3.1 which states that well-formed types of $\mathbf{Ob_{1<:}}$ get mapped to well-formed types of $\mathbf{F}_{<:\mu}$ and that the subtyping judgements of $\mathbf{Ob_{1<:}}$ are preserved by the translation.

$$\begin{array}{rcl} \phi^* & = & \phi \\ (E, x : A)^* & = & E^*, x : A^* \end{array}$$

Theorem 3.1

1. *If $E \vdash \diamond$ is derivable in $\mathbf{Ob_{1<:}}$, then $E^* \vdash \diamond$ is derivable in $\mathbf{F}_{<:\mu}$.*
2. *If $E \vdash A$ is derivable in $\mathbf{Ob_{1<:}}$, then $E^* \vdash A^*$ is derivable in $\mathbf{F}_{<:\mu}$.*
3. *If $E \vdash A{<:}B$ is derivable in $\mathbf{Ob_{1<:}}$, then $E^* \vdash A^*{<:}B^*$ is derivable in $\mathbf{F}_{<:\mu}$.*

The second part of the translation is for terms. To make the main ideas in the translation of terms transparent, we first informally explain the results of the translation as untyped λ-terms, omitting the type annotations associated with using recursive and existential types; we make the details precise later. Informally, every term a of $\mathbf{Ob_{1<:}}$ is mapped to its meaning, $\langle\!\langle a \rangle\!\rangle$, which is a λ-term. Apart from the typing restrictions imposed by the recursive and existential types, the translation of an object type is a record type with two fields l_i^{sel} and l_i^{upd} for each method l_i and a field *self*. The field l_i^{sel} is the method l_i treated as a function of self, and the field l_i^{upd} returns a new object when applied to a new method treated as a function of self. With this understanding, the translations of method invocation and update are straightforward.

$$\begin{array}{rcl} \langle\!\langle a.l_j \rangle\!\rangle & = & \langle\!\langle a \rangle\!\rangle \cdot l_j^{sel}(\langle\!\langle a \rangle\!\rangle \cdot \mathit{self}) \\ \langle\!\langle a.l_j \Leftarrow \varsigma(x)b \rangle\!\rangle & = & \langle\!\langle a \rangle\!\rangle \cdot l_j^{upd}(\lambda(x)\langle\!\langle b \rangle\!\rangle) \end{array}$$

The most delicate part of the translation of terms is that for objects. This may be expected since we did not do anything computationally interesting so far—we just delegated responsibility to the fields l_i^{sel} and l_i^{upd} provided by the interface of objects. To understand the translation of objects, it is instructive to consider first an incorrect attempt, which will also explain the presence of the field l_i^{upd}. Suppose we chose not to have the field l_i^{upd} in the record interface for objects and instead modeled method update by an update of the field l_i^{sel}, *i.e.*, for an object $o \equiv [l_i = \varsigma(x_i)b_i\ ^{i\in 1\ldots n}]$, we would have that

$$\langle\!\langle o.l_j \Leftarrow \varsigma(x)b \rangle\!\rangle \ = \ \langle\!\langle o \rangle\!\rangle \cdot l_j^{sel} := \lambda(x)\langle\!\langle b \rangle\!\rangle \qquad \text{(wrong)}$$

Invoking a method l_j of o would still be interpreted as extracting the l_j^{sel} field and applying it to the field *self*. Since the object o is a record with field l_i^{sel} equal to the method b_i treated as a function of its self parameter, and since method invocation is modeled by application to the field *self*, the field *self* then has to be the object itself. We arrive at the following recursive definition for $\langle\!\langle o \rangle\!\rangle$:

$$\langle\!\langle o \rangle\!\rangle \ = \ \{l_i^{sel} = \lambda(x_i)\langle\!\langle b_i \rangle\!\rangle\ ^{i\in 1\ldots n}, \mathit{self} = \langle\!\langle o \rangle\!\rangle\} \qquad \text{(wrong)}$$

The problem with this (functional) interpretation of objects is that when a method gets updated the object changes but since we only update the field l_j of the record, this change is not reflected in the field *self* and consequently we lose the dynamic binding of self. Thus, if some other method uses l_j in its body, then its invocation modeled by application

Table 2: Translation of $\mathbf{Ob_{1<:}}$ into $\mathbf{F}_{<:\mu}$

$$
\begin{array}{rcl}
\langle\!\langle x \rangle\!\rangle_E & = & x \\
\langle\!\langle [l_i = \varsigma(x_i : A) b_i \ ^{i\in 1\ldots n}] \rangle\!\rangle_E & = & \texttt{letrec } \mathit{create}\,(f_1 : A^* \to B_1^*)\ \ldots\ (f_n : A^* \to B_n^*) : A^* = \\
 & & \quad \texttt{fold}\ (A^*, \\
 & & \qquad \texttt{pack } X <: A^* = A^* \\
 & & \qquad \texttt{with } \{l_i^{sel} = f_i \ ^{i\in 1\ldots n}, \\
 & & \qquad\quad l_i^{upd} = \lambda(g : A^* \to B_i^*) \\
 & & \qquad\qquad \mathit{create}\,(f_1)\ \ldots\ (f_{i-1})\ (g)\ (f_{i+1})\ \ldots\ (f_n)\ ^{i\in 1\ldots n}, \\
 & & \qquad\quad \mathit{self} = \mathit{create}\,(f_1)\ \ldots\ (f_n)\} : C_A\{X\}) \\
 & & \quad \texttt{in } \mathit{create}\,(\lambda(x_1 : A^*)\langle\!\langle b_1 \rangle\!\rangle_{E, x_1 : A})\ \ldots\ (\lambda(x_n : A^*)\langle\!\langle b_n \rangle\!\rangle_{E, x_n : A}) \\
 & & \quad \text{where } A \equiv [l_i : B_i \ ^{i\in 1\ldots n}] \\
\langle\!\langle a.l \rangle\!\rangle_E & = & \texttt{open unfold}(\langle\!\langle a \rangle\!\rangle_E) \texttt{ as } X <: L_{l,B}, x : \{l^{sel} : (X \to B^*), \mathit{self} : X\} \\
 & & \texttt{in } (x \cdot l^{sel})(x \cdot \mathit{self}) : B^* \\
 & & \text{where } B = \langle E, a \rangle_l \\
\langle\!\langle a.l \Leftarrow \varsigma(x : A) b \rangle\!\rangle_E & = & \texttt{open unfold}(\langle\!\langle a \rangle\!\rangle_E) \texttt{ as } X <: A^*, y : C_A\{X\} \\
 & & \texttt{in } (y \cdot l^{upd})(\lambda(x : X)\langle\!\langle b \rangle\!\rangle_{E, x : A}) : A^*
\end{array}
$$

to the field *self* would not see the result of the update. So, an important idea in the context of our translation is that method update is not modeled as record update. The second idea to glean from this flawed attempt is that defining the object itself recursively would not reflect the computational behavior of objects accurately. Intuitively, update has no chance of working once the recursion freezes *self* to be the state of the object at the time of creation, *i.e.*, if recursion is used too soon. (Those familiar with the recursive-record interpretation [Car88] may note that the source of its problems in modeling method update can also be traced to the early use of recursion.)

The solution is to define not the object itself recursively, but the *dependence* of the object on its methods recursively. That is, we define a function *create* that when applied to n methods, returns an object with those n methods and it is the definition of *create* that is recursive. An object can then be defined by the application of *create* to its methods, as follows:

$$
\begin{array}{l}
\text{For } o \equiv [l_i = \varsigma(x_i) b_i \ ^{i\in 1\ldots n}], \\
\langle\!\langle o \rangle\!\rangle \quad = \quad \texttt{letrec } \mathit{create}\,(f_1) \cdots (f_n) = \\
\qquad \{l_i^{sel} = f_i \ ^{i\in 1\ldots n}, \\
\qquad\ l_i^{upd} = \lambda(g)\ \mathit{create}\,(f_1) \cdots (f_{i-1}) \\
\qquad\qquad\qquad (g)\ (f_{i+1}) \cdots (f_n)\ ^{i\in 1\ldots n}, \\
\qquad\ \mathit{self} = \mathit{create}\,(f_1) \cdots (f_n)\} \\
\qquad \texttt{in } \mathit{create}\,(\lambda(x_1)\langle\!\langle b_1 \rangle\!\rangle) \ldots (\lambda(x_n)\langle\!\langle b_n \rangle\!\rangle)
\end{array}
$$

We now define the translation of terms with typing annotations, more precisely. We use the following notation.

Notation:

1. For any object type $A \equiv [l_i : B_i \ ^{i\in 1\ldots n}]$, we define the $\mathbf{F}_{<:\mu}$ type $C_A\{X\}$ with free variable X:

$$
\begin{array}{rcl}
C_A\{X\} & \triangleq & \{l_i^{sel} : (X \to B_i^*) \ ^{i\in 1\ldots n}, \\
 & & \quad l_i^{upd} : (X \to B_i^*) \to X \ ^{i\in 1\ldots n}, \\
 & & \quad \mathit{self} : X\}
\end{array}
$$

2. For any method name l and $\mathbf{Ob_{1<:}}$ type B, we define the $\mathbf{F}_{<:\mu}$ type $L_{l,B}$ by:

$$
L_{l,B} \quad \triangleq \quad \mu(Y)\exists(X <: Y)\{l^{sel} : (X \to B^*), \mathit{self} : X\}
$$

3. Suppose, for any term a and environment E, that $E \vdash a : [\ldots, l : B, \ldots]$ is provable in $\mathbf{Ob_{1<:}}$. Then by the minimum-types property of $\mathbf{Ob_{1<:}}$ [AC94b] and by the invariance of object types, we have that if $E \vdash a : [\ldots, l : B', \ldots]$ then $B \equiv B'$. So we let $\langle E, a \rangle_l$ be the unique type B such that $E \vdash a : [\ldots, l : B, \ldots]$ is provable if it exists, and be undefined otherwise.

For any term a in $\mathbf{Ob_{1<:}}$ and environment E, Table 2 defines a term $\langle\!\langle a \rangle\!\rangle_E$ of $\mathbf{F}_{<:\mu}$. The translation proceeds by induction on the structure of a. In particular, the translation of a judgement $E \vdash a : A$ does not depend on its derivation in $\mathbf{Ob_{1<:}}$, and consequently, we can avoid coherence issues in our proofs. The inclusion of the environment E in defining the meaning of a term arises for purely technical reasons. It is to give the necessary type annotations in the translation of method invocation. If we had omitted type annotations from the target calculus or put more type information in the syntax of the term for method invocation, we could have defined the meaning of the term without any dependence on the environment.

Some remarks regarding the translation of terms are in order. The translation of method invocation explains the presence of the field *self* in the translation of object types: using x instead of $x \cdot \mathit{self}$ would not lead to a typable result. In the translation of method update, the use of $\lambda(x : X)\langle\!\langle b \rangle\!\rangle_{E, x : A}$ is motivated by the reduction rule (Eval Update) which asserts:

$$
([l_i = \varsigma(x_i : A) b_i \ ^{1\ldots n+m}].l_j \Leftarrow \varsigma(x : A') b) \ \rightsquigarrow \ [l_j = \varsigma(x : A) b, \ldots]
$$

with A instead of A' in the type annotation of x in the updated object. The use of $\lambda(x : A^*)\langle\!\langle b \rangle\!\rangle_{E, x : A}$ instead of $\lambda(x : X)\langle\!\langle b \rangle\!\rangle_{E, x : A}$ would be acceptable from the point of view of typing but would not fit with the rule (Eval Update).

The following theorem states that our translation preserves typing judgements and the computational behavior of terms.

Theorem 3.2

1. *If* $E \vdash a : A$ *is derivable in* $\mathbf{Ob_{1<:}}$ *then* $E^* \vdash \langle\langle a \rangle\rangle_E : A^*$ *is derivable in* $\mathbf{F}_{<:\mu}$.
2. *If* $E \vdash a : A$ *is derivable in* $\mathbf{Ob_{1<:}}$ *and* $a \twoheadrightarrow_o b$ *then* $\langle\langle a \rangle\rangle_E \twoheadrightarrow_f \langle\langle b \rangle\rangle_E$.

The translation can serve as a basis for validating reasoning principles for objects from reasoning principles for functions. In particular, we can prove that two objects are equivalent by showing that their translations are equivalent. We have been able to check a few non-trivial object equivalences in this manner. This proof method is not complete, because the translation is not fully abstract; however, it is sound, because the translation is computationally adequate, as we show next.

Let a and b be two closed $\mathbf{Ob_{1<:}}$ terms of type A. We say that a and b are operationally equivalent at type A, and write $a \equiv_o b : A$, if we have that $C[a] \Downarrow_o$ if and only if $C[b] \Downarrow_o$ for any context $C[\cdot]$ which is well-typed assuming the hole $[\cdot]$ is of type A. We define the relation of operational equivalence similarly for $\mathbf{F}_{<:\mu}$, and write $a \equiv_f b : A$. The first part of the following theorem states that the translation is computationally adequate; the second part, which is a corollary of the first, states that if two $\mathbf{Ob_{1<:}}$ terms have operationally equivalent translations then they are operationally equivalent.

Theorem 3.3 *Assume that* $\emptyset \vdash a : A$ *and* $\emptyset \vdash b : A$ *are derivable in* $\mathbf{Ob_{1<:}}$. *Then:*

1. $a \Downarrow_o$ *if and only if* $\langle\langle a \rangle\rangle_\emptyset \Downarrow_f$.
2. *If* $\langle\langle a \rangle\rangle_\emptyset \equiv_f \langle\langle b \rangle\rangle_\emptyset : A^*$ *then* $a \equiv_o b : A$.

In summary, there are three key ideas in the translation. The first is that interpreting an object type as a recursive type abstraction gives the desired subtypings. The second is to model method invocation not as application to the object itself, but rather to a field *self* which holds the current value of the object. And finally, by splitting each method into a field for invocation and a field for update and by using recursion in a function that creates objects, we obtain dynamic binding.

4 Interpretation of an Imperative Object Calculus

In this section, we show how the ideas embodied in the translation described in Section 3 are also useful to the interpretation of imperative object-oriented constructs. Our formal setting is the imperative object calculus of [AC95a].

4.1 An Imperative Object Calculus

The terms of the untyped imperative object calculus are generated by the grammar:

$$\begin{array}{rcl} a, b & ::= & x \mid [l_i = \varsigma(x_i)b_i \ ^{i \in 1 \ldots n}] \mid a.l \mid a.l \Leftarrow \varsigma(x)b \\ & & \mid \mathit{clone}(a) \mid \texttt{let } x = a \texttt{ in } b \end{array}$$

As the previous calculus, this imperative calculus has terms corresponding to objects, method invocation, and method update. However, the operational semantics is imperative in that method names denote locations where the closures of the corresponding methods are stored and method update is done in place. Thus, method update has a side-effect of changing the object rather than returning a new object. In addition, we have two new primitives: (1) *clone*(a) returns a shallow copy of the object a, *i.e.*, an object with the same method suite as a stored in fresh memory locations; (2) the let construct evaluates a term, binds it to a variable, and then evaluates a second term with that variable in scope. Sequential evaluation (;) and eagerly evaluated fields can be defined from `let`. The type system is given in Appendix C; it is an extension of that of $\mathbf{Ob_{1<:}}$.

4.2 Translation

We translate the imperative object calculus into an imperative version of $\mathbf{F}_{<:\mu}$, which here we describe informally. The syntax of this imperative version extends that of $\mathbf{F}_{<:\mu}$ with field update for records (written $a \cdot l := b$) and with an uninitialized value of each type (written $\mathit{nil}(B)$). Because of the presence of field update, record types must be invariant in their components. The operational semantics of the imperative version is significantly different from that of $\mathbf{F}_{<:\mu}$ in two respects: (1) The field names in records now denote memory locations and field update is done in place. It is therefore more accurate to think of a record as a collection of memory locations rather than as a collection of values. (2) In the presence of side-effects, one needs to fix an evaluation order; we assume call-by-value evaluation for the target calculus. Then `let` $x = a$ `in` b can be defined as $(\lambda(x)b)(a)$, and $a; b$ can be defined as $(\lambda(z)b)(a)$ for some z not free in b.

The main departure from the translation described in Section 3.3 is that, in the imperative setting, we do not split a method into two distinct fields corresponding to method invocation and method update. Recall that the essential reason for the split in the functional case was that the field *self* would not detect the change to a method if method update was modeled by record update of the field corresponding to the method. However, in the presence of imperative features in the target calculus, we can use the field *self* to store a pointer to the record itself (that is the meaning of the object), thus ensuring that any changes to the other fields of the record are reflected in *self*.

The translation of types uses the ideas described in Section 2. In addition, we include a cloning function in the public interface of an object. For types we therefore have:

$$\begin{array}{rcl} \mathit{Top}^* & = & \mathit{Top} \\ [l_i : B_i \ ^{i \in 1 \ldots n}]^* & = & \mu(Y)\exists(X <: Y) \\ & & \quad \{l_i : (X \to B_i^*) \ ^{i \in 1 \ldots n}, \\ & & \quad \mathit{clone} : \{\} \to X, \\ & & \quad \mathit{self} : X\} \end{array}$$

The distinction between the fields *self* and *clone* is that the former contains a pointer to the record itself while the latter returns a shallow copy of the record (under a dummy abstraction). We need to distinguish the two since, in method invocation, one must apply the method to the object rather than to a shallow copy of the object. As before, the use of a recursive type abstraction yields the desired subtypings.

Table 3: Translation of the Imperative Object Calculus (Sketch)

$$
\begin{array}{rcl}
\langle\langle x \rangle\rangle & = & x \\
\langle\langle [l_i = \varsigma(x_i) b_i \ ^{i \in 1 \ldots n}] \rangle\rangle & = & \texttt{letrec}\ \mathit{create}(f_1) \ldots (f_n) = \\
 & & \qquad \texttt{let}\ z = \{l_i = f_i \ ^{i \in 1 \ldots n},\ \mathit{clone} = \mathit{nil},\ \mathit{self} = \mathit{nil}\} \\
 & & \qquad \texttt{in}\ z \cdot \mathit{clone} := \lambda(x)\mathit{create}(z \cdot l_1) \cdots (z \cdot l_n); \\
 & & \qquad\quad z \cdot \mathit{self} := z; \\
 & & \qquad\quad z \cdot \mathit{self} \\
 & & \quad \texttt{in}\ \mathit{create}\ (\lambda(x_1)\langle\langle b_1 \rangle\rangle \ \ldots\ (\lambda(x_n)\langle\langle b_n \rangle\rangle) \\
\langle\langle a.l_j \rangle\rangle & = & \texttt{let}\ x = \langle\langle a \rangle\rangle\ \texttt{in}\ (x \cdot l_j)(x \cdot \mathit{self}) \\
\langle\langle a.l \Leftarrow \varsigma(x) b \rangle\rangle & = & \texttt{let}\ y = \langle\langle a \rangle\rangle\ \texttt{in}\ y \cdot l_j := \lambda(x)\langle\langle b \rangle\rangle \\
\langle\langle \mathit{clone}(a) \rangle\rangle & = & (\langle\langle a \rangle\rangle \cdot \mathit{clone})(\{\}) \\
\langle\langle \texttt{let}\ x = a\ \texttt{in}\ b \rangle\rangle & = & \texttt{let}\ x = \langle\langle a \rangle\rangle\ \texttt{in}\ \langle\langle b \rangle\rangle
\end{array}
$$

We give the precise definition of the translation of terms below. For now, we refer to Table 3 which states the translation omitting type annotations in terms. In the translation of an object, we declare a skeletal record structure z where the fields *clone* and *self* are uninitialized, and then update these fields so that they can point circularly to the record structure. Note that we retained the idea of defining a *create* function recursively rather than the object itself recursively. This is necessary for cloning to return the correct copy of the object after updates; if we had defined the object recursively, then clone would have been frozen to return a shallow copy of the state of the object at the time of its creation. (*Cf.* the semantics of [ESTZ95], which does not accommodate cloning.) In our translation, the field *clone* is defined to be an abstraction so that the application of *create* terminates under call-by-value evaluation. Method invocation is interpreted in the same way as in the functional case while method update is interpreted as record update. Cloning is interpreted as an application of the field *clone* to a dummy argument (the empty record).

The precise definition of the translation of terms, in Table 4, relies on the following notation:

Notation:

1. For $A \equiv [l_i : B_i \ ^{i \in 1 \ldots n}]$, we define:

$$
C_A^{imp}\{X\} \ \triangleq\ \{l_i : (X \to B_i^*) \ ^{i \in 1, \ldots, n},\ \mathit{clone} : \{\} \to X,\ \mathit{self} : X\}
$$

2. For any method name l and type B, we let:

$$
L_{l,B}^{imp} \ \triangleq\ \mu(Y)\exists(X <: Y)\{l : (X \to B^*), \mathit{self} : X\}
$$

3. The type $\mathit{MinTy}\langle E, a \rangle$ is the minimum type of a in environment E, *i.e.*, the type A such that $E \vdash a : A'$ is provable if and only if $E \vdash A <: A'$; it is undefined if a is not typable in E. The type $\langle E, a \rangle_l$ is as in Section 3.3.

We can prove a soundness theorem for this translation. We omit it from this paper since its statement requires lengthy definitions detailing and relating the operational semantics of the imperative calculi.

5 Extensions to Richer Object Types

In this section, we consider richer typing disciplines for objects: variance annotations, Self types, and structural rules (all described in [AC95b]); we show how our translation extends to account for them. In Section 5.1, we begin by giving an overview of these typing disciplines and an informal description of our interpretation for them. In Section 5.2, we describe an enriched object calculus more precisely. Finally, in Section 5.3, we give a translation of this object calculus.

5.1 Preview

Variance Annotations

Variance annotations are an extension to object types; they are symbols ($^+$, $^-$, ^{0}) attached to method names in object types. The annotation l^+ indicates that method l is only invocable, l^- indicates it is only updatable, and l^0 indicates that it is both. These annotations allow finer protection on the access of methods, and give desirable subtyping properties. Object types are covariant in the types of their $^+$ components, contravariant in the types of their $^-$ components, and invariant in the types of their ^{0} components.

Variance annotations naturally fit in the framework of our interpretation. Namely, we can translate object types to the same recursive type abstractions with both record components l^{sel}, l^{upd} for a method l^0; only the l^{sel} component for l^+; and only the l^{upd} component for l^-.

Self Types

The Self-type construct yields flexible typing for objects with methods that return objects of the type of self. Extending the notation for object types, we write

$$
\mathit{Obj}(X)[l_i : B_i\{X\} \ ^{i \in 1 \ldots n}]
$$

where Obj binds a type variable X that can occur covariantly in the result types B_i; intuitively the variable X stands for the type of self, called the Self type. A longer object type is still a subtype of a shorter one:

$$
\mathit{Obj}(X)[l_i : B_i \ ^{i \in 1 \ldots n+m}] <: \mathit{Obj}(X)[l_i : B_i \ ^{i \in 1 \ldots n}]
$$

Recall that in the translation of simple object types given by Equation 1 we viewed the representation type as the "true

Table 4: Translation of the Imperative Object Calculus

$$
\begin{array}{rcl}
\langle\!\langle x \rangle\!\rangle_E & = & x \\
\langle\!\langle [l_i = \varsigma(x_i : A) b_i \ ^{i \in 1 \ldots n}] \rangle\!\rangle_E & = & \texttt{letrec } \mathit{create}(f_1 : A^* \to B_1^*) \ldots (f_n : A^* \to B_n^*) : A^* = \\
& & \quad \texttt{let } z : C_A^{imp}\{A^*\} = \{l_i = f_i \ ^{i \in 1 \ldots n},\ \mathit{clone} = \mathit{nil}(\{\} \to A^*),\ \mathit{self} = \mathit{nil}(A^*)\} \\
& & \quad \texttt{in } z \cdot \mathit{clone} := \lambda(x : \{\})\, \mathit{create}\,(z \cdot l_1) \cdots (z \cdot l_n); \\
& & \qquad z \cdot \mathit{self} := \texttt{fold}(A^*, \texttt{pack } X <: A^* = A^* \texttt{ with } z : C_A^{imp}\{X\}); \\
& & \qquad z \cdot \mathit{self} \\
& & \quad \texttt{in } \mathit{create}\ (\lambda(x_1 : A^*)\langle\!\langle b_1 \rangle\!\rangle_{E, x_1 : A}) \ \ldots\ (\lambda(x_n : A^*)\langle\!\langle b_n \rangle\!\rangle_{E, x_n : A}) \\
& & \quad \text{where } A \equiv [l_i : B_i \ ^{i \in 1 \ldots n}] \\
\langle\!\langle a.l \rangle\!\rangle_E & = & \texttt{open unfold}(\langle\!\langle a \rangle\!\rangle_E) \texttt{ as } X <: L_{l,B}^{imp}, x : \{l : (X \to B^*), \mathit{self} : X\} \\
& & \texttt{in } (x \cdot l)\,(x \cdot \mathit{self}) : B^* \\
& & \text{where } B = \langle E, a \rangle_l \\
\langle\!\langle a.l \Leftarrow \varsigma(x : A) b \rangle\!\rangle_E & = & \texttt{open unfold}(\langle\!\langle a \rangle\!\rangle_E) \texttt{ as } X <: A^*, y : C_A^{imp}\{X\} \\
& & \texttt{in fold}(A^*, \\
& & \qquad \texttt{pack } X' <: A^* = X \\
& & \qquad \texttt{with } y \cdot l_j := \lambda(x : X)\langle\!\langle b \rangle\!\rangle_{E, x : A} : C_A^{imp}\{X'\}) : A^* \\
\langle\!\langle \mathit{clone}(a) \rangle\!\rangle_E & = & \texttt{open unfold}(\langle\!\langle a \rangle\!\rangle_E) \texttt{ as } X <: A^*, x : C_A^{imp}\{X\} \\
& & \texttt{in } (x \cdot \mathit{clone})(\{\}) \\
& & \text{where } A = \mathit{MinTy}\langle E, a \rangle \\
\langle\!\langle \texttt{let } x : A = a \texttt{ in } b \rangle\!\rangle_E & = & \texttt{let } x : A^* = \langle\!\langle a \rangle\!\rangle_E \texttt{ in } \langle\!\langle b \rangle\!\rangle_{E, x : A}
\end{array}
$$

type" of an object. We take this "true type" to be the Self type; therefore, for $A \equiv Obj(X)[l_i : B_i\{X\}\ ^{i \in 1 \ldots n}]$, we let:

$$
\begin{array}{rcl}
A^* & = & \exists(X <: A^*)\{l_i^{sel} : X \to B_i^*\{X\}\ ^{i \in 1 \ldots n}, \\
& & \qquad l_i^{upd} : (X \to B_i^*\{X\}) \to X\ ^{i \in 1 \ldots n}, \\
& & \qquad \mathit{self} : X\}
\end{array}
$$

With this straightforward extension, our interpretation accounts for Self types.

Structural Rules

While the subtyping rules for object types assert that a longer object type is a subtype of a shorter object type, structural rules arise as consequence of the stronger "structural assumption" that the only subtypes of an object type are longer object types. An example of such a structural rule, using the simple object types of $\mathbf{Ob_{1<:}}$, is the following modification of the rule (Val Update) of Table 8:

(Struct Val Update)
For $A \equiv [l_i : B_i\ ^{i \in 1 \ldots n}]$

$$\frac{E \vdash C <: A \quad E \vdash a : C \quad E, x : C \vdash b : B_j}{E \vdash a.l_j \Leftarrow \varsigma(x : C) b : C}$$

In our interpretation, structural assumptions on object types are reflected as structural assumptions on recursive types. Specifically, structural rules for object types are validated if we strengthen the target calculus with a structural rule for recursive types:

(Struct Val Unfold)

$$\frac{E \vdash C <: \mu(X) B\{X\} \quad E \vdash a : C}{E \vdash \texttt{unfold}(a) : B\{C\}}$$

The rule (Struct Val Unfold) can be seen as a consequence of assuming that any subtype of a recursive type arises through the reflexivity rule ((Sub Refl) of Table 7) or the subtyping rule for recursive types ((Sub Rec) of Table 9). For example, suppose that $E \vdash C <: \mu(X)B\{X\}$ because of (Sub Rec). Then C is of the form $\mu(X)B'\{X\}$ and if $E \vdash a : C$ then $E \vdash \texttt{unfold}(a) : B'\{C\}$. Further, we have that $E, Y <: Top, X <: Y \vdash B'\{X\} <: B\{Y\}$. In particular, since $E \vdash C <: C$, using C for both X and Y we get that $E \vdash B'\{C\} <: B\{C\}$ and using subsumption we get the consequent of the rule (Struct Val Unfold).

We can see informally how the rule (Struct Val Update) is validated thanks to (Struct Val Unfold). Assume that $E \vdash C^* <: A^*$ and $E \vdash \langle\!\langle a \rangle\!\rangle : C^*$. Using the definition of A^* as a recursive type and applying (Struct Val Unfold), we can conclude that:

$$E \vdash \texttt{unfold}(\langle\!\langle a \rangle\!\rangle) : \exists(X <: C^*)\{\ldots, l_j^{upd} : (\cdots) \to X, \ldots\}$$

The result of an update is of type C^*, since l_j^{upd} returns a result of type X and $X <: C^*$. In contrast, with the weaker, non-structural rule (Val Unfold) of $\mathbf{F}_{<:\mu}$ (Table 9), we can conclude only that $E \vdash \texttt{unfold}(\langle\!\langle a \rangle\!\rangle) : \exists(X <: A^*)\{\ldots\}$ and the result of the update has to be given the weaker type A^*.

5.2 An Enriched Object Calculus

The calculus $\mathbf{Ob}_{<:}^{str}$ is an extension of $\mathbf{Ob_{1<:}}$ with variance annotations, Self types, and structural rules. Like the semantics of $\mathbf{Ob_{1<:}}$, the semantics of $\mathbf{Ob}_{<:}^{str}$ is free of side-effects.

The types of $\mathbf{Ob}_{<:}^{str}$ are generated by the grammar:

$$A, B \ ::= \ X \mid \mathit{Top} \mid \mathit{Obj}(X)[l_i \nu_i : B_i\{X\}\ ^{i \in 1 \ldots n}]$$

where $\nu_i \in \{^+, ^-, ^0\}$. As described in Section 5.1, *Obj* binds the Self type, and the variance annotation ν_i specifies the operations permissible on method l_i.

Table 5: Operational Semantics of $\mathbf{Ob}^{str}_{<:}$

If $a \equiv obj(X=A)[l_i = \varsigma(x_i:X)b_i\{X,x_i\}^{\ i\in 1\ldots n}]$, $j \in 1\ldots n$			
(Eval Select)	$a.l_j$	$\rightsquigarrow$	$b_j\{A,a\}$
(Eval Update)	$a.l_j \Leftarrow (Y<:A', y:Y)\varsigma(x:Y)b\{Y,y\}$	$\rightsquigarrow$	$obj(X=A)[l_j = \varsigma(x:X)b\{X,a\},$ $l_i = \varsigma(x_i:X)b_i^{\ i\in\{1\ldots n\}-\{j\}}]$

Because of Self types, the term syntax of $\mathbf{Ob}^{str}_{<:}$ is slightly different from that of $\mathbf{Ob}_{1<:}$:

$$\begin{array}{rcl} a,b & ::= & x \mid obj(X=A)[l_i = \varsigma(x_i:X)b_i^{\ i\in 1\ldots n}] \\ & & \mid a.l \mid a.l \Leftarrow (Y<:A, y:Y)\varsigma(x:Y)b \end{array}$$

An object has the form $obj(X=A)[l_i = \varsigma(x_i:X)b_i^{\ i\in 1\ldots n}]$ with X standing for the Self type. Method update is written $a.l \Leftarrow (Y<:A, y:Y)\varsigma(x:Y)b$ where A is a known type for a, Y denotes the Self type of a, y is bound to the object being updated (a), and x is the usual self parameter in method b. The parameter y is useful because it is given type Y while a has the weaker type A.

The typing rules for $\mathbf{Ob}^{str}_{<:}$ are structural. They appear in Appendix D.

The operational semantics is defined via a reduction system whose redexes are given in Table 5. In the rule for method update, note that the object a gets substituted for the parameter y. Apart from this, the only difference from the corresponding rules of $\mathbf{Ob}_{1<:}$ is the type propagation—the actual type of self gets substituted for the formal type parameter X standing for the Self type. We denote the many-step reduction relation for $\mathbf{Ob}^{str}_{<:}$ by $\longrightarrow\!\!\!\!\!\rightarrow_{os}$.

5.3 Translation

We translate $\mathbf{Ob}^{str}_{<:}$ into an extension $\mathbf{F}^{str}_{<:\mu}$ of $\mathbf{F}_{<:\mu}$; this extension has the same operational semantics as $\mathbf{F}_{<:\mu}$ but includes a structural rule, namely the rule (Struct Val Unfold) of Section 5.1.

The translation of types combines the ideas for variance annotations and for Self types described in Section 5.1:

$$\begin{array}{rcl} X^* & = & X \\ Top^* & = & Top \\ Obj(X)[l_i\nu_i : B_i\{X\}^{\ i\in 1\ldots n}]^* & & \\ \quad = & \mu(Y)\exists(X<:Y)\{(l_i\nu_i : B_i\{X\})^{\dagger\ i\in 1\ldots n}, & \\ & \quad self : X\} & \\ & \text{for } Y \text{ not free in the } B_i\text{'s} & \end{array}$$

where the fields $(l_i\nu_i : B_i\{X\})^\dagger$ are defined by case analysis on the variance annotation ν_i as follows:

$$\begin{array}{rcl} (l_i^+ : B_i\{X\})^\dagger & = & l_i^{sel} : X \to B_i^*\{X\} \\ (l_i^- : B_i\{X\})^\dagger & = & l_i^{upd} : (X \to B_i^*\{X\}) \to X \\ (l_i^0 : B_i\{X\})^\dagger & = & (l_i^+ : B_i\{X\})^\dagger, (l_i^- : B_i\{X\})^\dagger \end{array}$$

We define the translation of environments as in Section 3.3, with the additional clause $(E, X<:A)^* = E^*, X<:A^*$.

The following theorem states that well-formed environments are mapped to well-formed environments, that well-formed types are mapped to well-formed types, and that the translation preserves subtyping judgements.

Theorem 5.1

1. *If $E \vdash \diamond$ is derivable in $\mathbf{Ob}^{str}_{<:}$, then $E^* \vdash \diamond$ is derivable in $\mathbf{F}_{<:\mu}$, and a fortiori in $\mathbf{F}^{str}_{<:\mu}$.*
2. *If $E \vdash A$ is derivable in $\mathbf{Ob}^{str}_{<:}$, then $E^* \vdash A^*$ is derivable in $\mathbf{F}_{<:\mu}$, and a fortiori in $\mathbf{F}^{str}_{<:\mu}$.*
3. *If $E \vdash A<:B$ is derivable in $\mathbf{Ob}^{str}_{<:}$, then $E^* \vdash A^*<:B^*$ is derivable in $\mathbf{F}_{<:\mu}$, and a fortiori in $\mathbf{F}^{str}_{<:\mu}$.*

We give the translation of terms in Table 6, using the following notation:

Notation:

1. For $A \equiv Obj(X)[l_i\nu_i : B_i\{X\}]$, we define:

$$C^{str}_A\{X\} \triangleq \{(l_i\nu_i : B_i\{X\})^{\dagger\ i\in 1\ldots n}, self : X\}$$

2. For a type A, environment E, and method name l, we define the $\mathbf{Ob}^{str}_{<:}$ type $\langle A,E\rangle_l$ as follows. If $A \equiv Obj(X)[\ldots, l\nu : B\{X\}, \ldots]$, then $\langle A,E\rangle_l$ is $B\{X\}$. If $A \equiv X$ (a type variable) and $E \equiv E', X<:A', E''$, then $\langle A,E\rangle_l$ is $\langle A',E'\rangle_l$. In all other cases (e.g., for $A \equiv Top$), $\langle A,E\rangle_l$ is undefined.
3. As in Section 4.2, the type $MinTy\langle E,a\rangle$ is the minimum type of a in environment E. (We can prove that such a minimum type exists in $\mathbf{Ob}^{str}_{<:}$.)

If we omit type annotations then the translation of terms is basically the same as that described for $\mathbf{Ob}_{1<:}$. The main novelty of the translation is that it shows that we can attach suitable type annotations to the untyped terms described in Section 3.3 so that well-typed terms of $\mathbf{Ob}^{str}_{<:}$ get mapped to well-typed terms of $\mathbf{F}^{str}_{<:\mu}$. The following theorem states that the translation preserves typing judgements and computational behavior.

Theorem 5.2

1. *If $E \vdash a : A$ is derivable in $\mathbf{Ob}^{str}_{<:}$ then $E^* \vdash \langle\langle a\rangle\rangle_E : A^*$ is derivable in $\mathbf{F}^{str}_{<:\mu}$.*
2. *If $E \vdash a : A$ is derivable in $\mathbf{Ob}^{str}_{<:}$ and $a \longrightarrow\!\!\!\!\!\rightarrow_{os} b$ then $\langle\langle a\rangle\rangle_E \longrightarrow\!\!\!\!\!\rightarrow_f \langle\langle b\rangle\rangle_E$.*

Table 6: Translation of $\mathbf{Ob}^{str}_{<:}$ into $\mathbf{F}^{str}_{<:\mu}$

$$
\begin{array}{rcl}
\langle\!\langle x \rangle\!\rangle_E & = & x \\
\langle\!\langle\, obj(X=A)[l_i = \varsigma(x_i : X) b_i\{X\}^{\ i\in 1\ldots n}] \rangle\!\rangle_E & & \\
 & = & \texttt{letrec}\ create(f_1 : A^* \to B_1^*\{A^*\}) \ \ldots\ (f_n : A^* \to B_n^*\{A^*\}) : A^* = \\
 & & \quad \texttt{fold}\,(A^*, \\
 & & \qquad \texttt{pack}\ X <: A^* = A^* \\
 & & \qquad \texttt{with}\,\{l_i^{sel} = f_i\ ^{i\in 1\ldots n}, \\
 & & \qquad\quad l_i^{upd} = \lambda(g : A^* \to B_i^*\{A^*\}) \\
 & & \qquad\qquad\qquad create\ (f_1)\ \ldots\ (f_{i-1})\ (g)\ (f_{i+1})\ \ldots\ (f_n)\ ^{i\in 1\ldots n}, \\
 & & \qquad\quad self = create(f_1)\ \ldots\ (f_n)\} : C_A^{str}\{X\}) \\
 & & \quad \texttt{in}\ create\ (\lambda(x_1 : A^*)\langle\!\langle b_1\{A\}\rangle\!\rangle_{E, x_1 : A})\ \ldots\ (\lambda(x_n : A^*)\langle\!\langle b_n\{A\}\rangle\!\rangle_{E, x_n : A}) \\
 & & \quad \text{where}\ A \equiv Obj(X)[l_i \nu_i : B_i\{X\}^{\ i\in 1\ldots n}] \\
\langle\!\langle a.l \rangle\!\rangle_E & = & \texttt{open unfold}(\langle\!\langle a \rangle\!\rangle_E)\ \texttt{as}\ X <: A^*, x : \{l^{sel} : (X \to B^*\{X\}), self : X\} \\
 & & \texttt{in}\ (x \cdot l^{sel})(x \cdot self) : B^*\{A^*\} \\
 & & \text{where}\ A = MinTy\langle E, a\rangle, B\{X\} = \langle A, E\rangle_l \\
\langle\!\langle a.l \Leftarrow (Y <: A, y : Y)\varsigma(x : Y) b \rangle\!\rangle_E & = & \texttt{open unfold}(\langle\!\langle a \rangle\!\rangle_E)\ \texttt{as}\ X <: A^*, z : \{l^{upd} : (X \to B^*\{X\}) \to X, self : X\} \\
 & & \texttt{in}\ (z \cdot l^{upd})((\lambda(Y <: A^*)\lambda(y : Y)\lambda(x : Y)\langle\!\langle b \rangle\!\rangle_{E, Y <: A, y : Y, x : Y})\ X\ (z \cdot self)) : A^* \\
 & & \text{where}\ B = \langle A, E\rangle_l
\end{array}
$$

6 Conclusions

We have presented a new interpretation of objects and object types that preserves subtyping and behavior; its basic idea works for both functional and imperative semantics. Our interpretation is more general than previous solutions in that it handles object-based constructs such as cloning and method update, as well as the common class-based constructs. Moreover, it is simpler than other proposals in the sense of being syntax-directed. It is the first interpretation of this kind.

Our interpretation offers insights into the nature of objects. It describes, in principle, a type-safe way of coding objects in procedural languages. However, as is the case even with more limited interpretations, it cannot be used in actual programming practice because of its pragmatic complexity. This fact confirms the commonly held belief that object-oriented languages differ significantly from procedural languages in practical expressive power.

References

[AC94a] M. Abadi and Luca Cardelli. A semantics of object types. In *Proceedings of the Ninth Annual Symposium on Logic in Computer Science*, pages 332–341, July 1994.

[AC94b] M. Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. In *Theoretical Aspects of Computer Software*, pages 296–320. Springer-Verlag, April 1994.

[AC95a] M. Abadi and L. Cardelli. An imperative object calculus: Basic typing and soundness. In *SIPL '95 — Proceedings of the Second ACM SIGPLAN Workshop on State in Programming Languages*. Technical Report UIUCDCS-R-95-1900, Department of Computer Science, University of Illinois at Urbana-Champaign, January 1995.

[AC95b] Martin Abadi and Luca Cardelli. An imperative object calculus. In P.D. Mosses, M. Nielsen, and M.I. Schwartzbach, editors, *TAPSOFT'95: Theory and Practice of Software Development*, pages 471–485. Springer-Verlag LNCS 915, May 1995.

[ALBC+93] O. Agesen, C. Chambers L. Bak, B.W. Chang, U. Holzle, J. Maloney, R.B. Smith, D. Ungar, and M. Wolczko. *The Self 3.0 programmer's reference manual*. Sun Microsystems, 1993.

[And92] B. Andersen. Ellie: a general, fine-grained, first-class, object-based language. *Journal of Object Oriented Programming*, 5(2):35–42, 1992.

[App93] Apple Computer, Inc. *Apple, The NewtonScript Programming Language*, 1993.

[Car88] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988. Special issue devoted to *Symp. on Semantics of Data Types*, Sophia-Antipolis (France), 1984.

[Car95] L. Cardelli. A language with distributed scope. In *Conference Record of the Twenty-Second Annual ACM Symposium on Principles of Programming Languages*, 1995.

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

[ESTZ95] J. Eifrig, S. Smith, V. Trifonov, and A. Zwarico. An interpretation of typed OOP in a language with state. *Lisp and Symbolic Computation*, 1995. To appear.

[HP95] Martin Hofmann and Benjamin Pierce. A unifying type-theoretic framework for objects. *Journal of Functional Programming*, 1995. To appear. Previous version appeared in the Symposium on Theoretical Aspects of Computer Science, 1994 (pages 251–262).

[Kam88] S. Kamin. Inheritance in Smalltalk-80: a denotational definition. In *ACM Symp. Principles of Programming Languages*, pages 80–87, 1988.

[Lie81] H. Lieberman. A preview of Act1. Technical Report AI Memo No 625, MIT, 1981.

[MGV92] B.A. Myers, D.A. Giuse, and B. Vander Zanden. Declarative programming in a prototype-instance

system: object-oriented programming without writing methods. In *Proc. OOPSLA '92*, pages 184–200, 1992.

[MMPN93] O.L. Madsen, B. Moller-Pedersen, and K. Nygaard. *Object-oriented programming in the Beta programming language*. Addison-Wesley, 1993.

[MP88] J.C. Mitchell and G.D. Plotkin. Abstract types have existential types. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, 1988. Preliminary version appeared in *Proc. 12th ACM Symp. on Principles of Programming Languages*, 1985.

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

[Rém94] D. Rémy. Programming Objects with ML-ART, an extension to ML with Abstract and Record types. In *Theoretical Aspects of Computer Software*. Springer-Verlag, April 1994.

[Tai92] A. Taivalsaari. Kevo, a prototype-based object-oriented language based on concatenation and module operations. Technical Report LACIR 92-02, University of Victoria, 1992.

Appendix

In this appendix we summarize several calculi, giving both grammars and rules. We often use assertions of the form $E \vdash \mathcal{J}_i \;\; \forall i \in 1 \ldots n$ to indicate n hypotheses; by convention, this means $E \vdash \diamond$ when $n = 0$.

A The $\mathbf{Ob}_{1<:}$ Calculus

The calculus $\mathbf{Ob}_{1<:}$ consists of the rules given in Tables 7 and 8. It has the following syntax:

Environments	$E \;::=\; \emptyset \mid E, x : A$
Types	$A, B \;::=\; \mathit{Top} \mid [l_i : B_i \;^{i \in 1 \ldots n}]$
Variables	x, y
Terms	$a, b \;::=\; x \mid [l_i = \varsigma(x_i : A) b_i \;^{i \in 1 \ldots n}]$ $\mid a.l \mid a.l \Leftarrow \varsigma(x : A) b$
Results	$v \;::=\; [l_i = \varsigma(x_i : A) b_i \;^{i \in 1 \ldots n}]$

B The $\mathbf{F}_{<:\mu}$ Calculus

The calculus $\mathbf{F}_{<:\mu}$ consists of the rules given in Tables 7 and 9. It has the following syntax:

Environments	$E \;::=\; \emptyset \mid E, x : A \mid E, X{<:}A$
Type Variables	X, Y
Types	$A, B, C \;::=\; X \mid \mathit{Top} \mid A \to B$ $\mid \{l_i : B_i \;^{i \in 1 \ldots n}\} \mid \mu(X) A$ $\mid \forall(X{<:}A) B \mid \exists(X{<:}A) B$
Variables	x, y
Terms	$a, b, c, d \;::=\; x \mid \lambda(x : A) b \mid a(b)$ $\mid \{l_i = b_i \;^{i \in 1 \ldots n}\} \mid a \cdot l$ $\mid \mathtt{fold}(A, b) \mid \mathtt{unfold}(a)$ $\mid \lambda(X{<:}A) b \mid b(A)$ $\mid \mathtt{pack}\ X{<:}A = C\ \mathtt{with}\ b : B\{X\}$ $\mid \mathtt{open}\ c\ \mathtt{as}\ X{<:}A, x : B\ \mathtt{in}\ d : D$
Results	$v \;::=\; \lambda(x : A) b \mid \{l_i = b_i \;^{i \in 1 \ldots n}\}$ $\mid \mathtt{fold}(A, v) \mid \lambda(X{<:}A) b$ $\mid \mathtt{pack}\ X{<:}A = C\ \mathtt{with}\ b : B\{X\}$

Other definitions of the set of results could be adopted. The one given here is convenient for our adequacy theorem; it is however not particularly compelling. Fortunately our techniques are not too sensitive to changes in the definition of the set of results.

C The Imperative Variant of $\mathbf{Ob}_{1<:}$

The typed imperative object calculus contains all the rules of $\mathbf{Ob}_{1<:}$ (described in Appendix A) and contains the typing rules given in Table 10 for its additional terms. As for $\mathbf{Ob}_{1<:}$, we can prove a minimum-types property for the typed imperative calculus. (This is a convenient departure from the original calculus of [AC95a]: the terms described here contain more type information.) The syntax is:

Environments	$E \;::=\; \emptyset \mid E, x : A$
Types	$A, B \;::=\; \mathit{Top} \mid [l_i : B_i \;^{i \in 1 \ldots n}]$
Variables	x, y
Terms	$a, b \;::=\; x \mid [l_i = \varsigma(x_i : A) b_i \;^{i \in 1 \ldots n}]$ $\mid a.l \mid a.l \Leftarrow \varsigma(x : A) b$ $\mid \mathit{clone}(a)$ $\mid \mathtt{let}\ x : A = a\ \mathtt{in}\ b$

D The $\mathbf{Ob}^{str}_{<:}$ Calculus

The calculus $\mathbf{Ob}^{str}_{<:}$ consists of the rules given in Table 7, the rules (Env X), (Type X), (Sub X) given in Table 9, and the rules of Table 11. It has the following syntax:

Environments	$E \;::=\; \emptyset \mid E, x : A \mid E, X{<:}A$
Type Variables	X, Y
Types	$A, B \;::=\; X \mid \mathit{Top}$ $\mid \mathit{Obj}(X)[l_i \nu_i : B_i \;^{i \in 1 \ldots n}]$ with $\nu_i \in \{^+, ^-, ^0\}$
Variables	x, y
Terms	$a, b \;::=\; x$ $\mid \mathit{obj}(X{=}A)[l_i {=} \varsigma(x_i : X) b_i \;^{i \in 1 \ldots n}]$ $\mid a.l$ $\mid a.l \Leftarrow (Y{<:}A, y : Y)\varsigma(x : Y) b$

Table 7: Common Typing Rules

Environments

(Env $\emptyset$) $\dfrac{}{\emptyset \vdash \diamond}$ (Env x) $\dfrac{E \vdash A}{E, x : A \vdash \diamond}$, $x \notin dom(E)$

Subtyping

(Sub Refl) $\dfrac{E \vdash A}{E \vdash A{<:}A}$ (Sub Trans) $\dfrac{E \vdash A{<:}B \quad E \vdash B{<:}C}{E \vdash A{<:}C}$

(Val Subsmp) $\dfrac{E \vdash a : A \quad E \vdash A{<:}B}{E \vdash a : B}$

Top

(Type Top) $\dfrac{E \vdash \diamond}{E \vdash \mathit{Top}}$ (Sub Top) $\dfrac{E \vdash A}{E \vdash A{<:}\mathit{Top}}$

Variable Typing

(Val x) $\dfrac{E', x : A, E'' \vdash \diamond}{E', x : A, E'' \vdash x : A}$

Table 8: Additional Typing Rules for $\mathbf{Ob_{1<:}}$

Object Types and Subtyping

(Type Object) $\dfrac{E \vdash B_i \quad \forall i \in 1 \ldots n}{E \vdash [l_i : B_i \ ^{i \in 1 \ldots n}]}$ (Sub Object) $\dfrac{E \vdash B_i \quad \forall i \in 1 \ldots n+m}{E \vdash [l_i : B_i \ ^{i \in 1 \ldots n+m}]{<:}[l_i : B_i \ ^{i \in 1 \ldots n}]}$

Term Typings

(Val Object) $\dfrac{E, x_i : A \vdash b_i : B_i \quad \forall i \in 1 \ldots n}{E \vdash [l_i = \varsigma(x_i : A) b_i \ ^{i \in 1 \ldots n}] : A}$, $A \equiv [l_i : B_i \ ^{i \in 1 \ldots n}]$

(Val Select) $\dfrac{E \vdash a : [l_i : B_i \ ^{i \in 1 \ldots n}]}{E \vdash a.l_j : B_j}$, $j \in 1 \ldots n$

(Val Update) $\dfrac{E \vdash a : A \quad E, x : A \vdash b : B_j}{E \vdash a.l_j \Leftarrow \varsigma(x : A) b : A}$, $A \equiv [l_i : B_i \ ^{i \in 1 \ldots n}]$, $j \in 1 \ldots n$

Table 9: Additional Typing Rules for $\mathbf{F}_{<:\mu}$

Environments

(Env X) $\dfrac{E \vdash A}{E, X{<:}A \vdash \diamond}, \quad X \notin dom(E)$

Types

(Type X) $\dfrac{E', X{<:}A, E'' \vdash \diamond}{E', X{<:}A, E'' \vdash X}$ (Type $\to$) $\dfrac{E \vdash A \quad E \vdash B}{E \vdash A{\to}B}$

(Type Record) $\dfrac{E \vdash B_i \quad \forall i \in 1 \ldots n}{E \vdash \{l_i : B_i \ ^{i \in 1 \ldots n}\}}$ (Type Rec) $\dfrac{E, X{<:}\mathit{Top} \vdash A}{E \vdash \mu(X)A}$

(Type All) $\dfrac{E, X{<:}A \vdash B}{E \vdash \forall(X{<:}A)B}$ (Type Exists) $\dfrac{E, X{<:}A \vdash B}{E \vdash \exists(X{<:}A)B}$

Subtyping

(Sub X) $\dfrac{E', X{<:}A, E'' \vdash \diamond}{E', X{<:}A, E'' \vdash X{<:}A}$ (Sub $\to$) $\dfrac{E \vdash A'{<:}A \quad E \vdash B{<:}B'}{E \vdash A{\to}B{<:}A'{\to}B'}$

(Sub Record) $\dfrac{E \vdash B_i{<:}B'_i, \ \forall i \in 1 \ldots n \qquad E \vdash B_i, \quad \forall i \in n+1 \ldots n+m}{E \vdash \{l_i : B_i \ ^{i \in 1 \ldots n+m}\}{<:}\{l_i : B'_i \ ^{i \in 1 \ldots n}\}}$

(Sub Rec) $\dfrac{E \vdash \mu(X)A \qquad E \vdash \mu(Y)B \qquad E, Y{<:}\mathit{Top}, X{<:}Y \vdash A{<:}B}{E \vdash \mu(X)A{<:}\mu(Y)B}$

(Sub All) $\dfrac{E \vdash A'{<:}A \quad E, X{<:}A' \vdash B{<:}B'}{E \vdash \forall(X{<:}A)B{<:}\forall(X{<:}A')B'}$ (Sub Exists) $\dfrac{E \vdash A{<:}A' \quad E, X{<:}A \vdash B{<:}B'}{E \vdash \exists(X{<:}A)B{<:}\exists(X{<:}A')B'}$

Term Typings

(Val Fun) $\dfrac{E, x : A \vdash b : B}{E \vdash \lambda(x : A)b : A{\to}B}$ (Val Appl) $\dfrac{E \vdash b : A{\to}B \qquad E \vdash a : A}{E \vdash b(a) : B}$

(Val Record) $\dfrac{E \vdash b_i : B_i \quad \forall i \in 1 \ldots n}{E \vdash \{l_i = b_i \ ^{i \in 1 \ldots n}\} : \{l_i : B_i \ ^{i \in 1 \ldots n}\}}$

(Val Record Select) $\dfrac{E \vdash a : \{l_i : B_i \ ^{i \in 1 \ldots n}\}}{E \vdash a \cdot l_j : B_j}, \quad j \in 1 \ldots n$

(Val Fold) $\dfrac{E \vdash b : B\{A\}}{E \vdash \mathtt{fold}(A, b) : A}, \quad A \equiv \mu(X)B\{X\}$

(Val Unfold) $\dfrac{E \vdash b : A}{E \vdash \mathtt{unfold}(b) : B\{A\}}, \quad A \equiv \mu(X)B\{X\}$

(Val Fun2) $\dfrac{E, X{<:}A \vdash b : B}{E \vdash \lambda(X{<:}A)b : \forall(X{<:}A)B}$ (Val Appl2) $\dfrac{E \vdash b : \forall(X{<:}A)B \quad E \vdash A'{<:}A}{E \vdash b(A') : B\{A'\}}$

(Val Pack) $\dfrac{E \vdash C{<:}A \qquad E \vdash b\{C\} : B\{C\}}{E \vdash \mathtt{pack}\ X{<:}A = C\ \mathtt{with}\ b\{X\} : B\{X\} : \exists(X{<:}A)B\{X\}}$

(Val Open) $\dfrac{E \vdash c : \exists(X{<:}A)B \quad E \vdash D \quad E, X{<:}A, x : B \vdash d : D}{E \vdash (\mathtt{open}\ c\ \mathtt{as}\ X{<:}A, x : B\ \mathtt{in}\ d : D) : D}$

Table 10: Additional Typing Rules for the Imperative Calculus

(Val Clone) $\dfrac{E \vdash a : [l_i : B_i \ ^{i \in 1 \ldots n}]}{E \vdash \mathit{clone}(a) : [l_i : B_i \ ^{i \in 1 \ldots n}]}$

(Val Let) $\dfrac{E \vdash a : A \quad E, x : A \vdash b : B}{E \vdash \mathtt{let}\ x : A = a\ \mathtt{in}\ b : B}$

Table 11: Additional Typing Rules for $\mathbf{Ob}^{str}_{<:}$

Variance Subtypings

(Sub Covariant) $\dfrac{E \vdash B <: B' \quad \nu \in \{^0, ^+\}}{E \vdash \nu B <: ^+ B'}$ (Sub Contravariant) $\dfrac{E \vdash B' <: B \quad \nu \in \{^0, ^-\}}{E \vdash \nu B <: ^- B'}$

(Sub Invariant) $\dfrac{E \vdash B}{E \vdash {}^0 B <: {}^0 B}$

Object Types and Subtyping

(Type Object) $\dfrac{E, X <: Top \vdash B_i\{X\} \quad \forall i \in 1 \ldots n}{E \vdash Obj(X)[l_i \nu_i : B_i \ ^{i \in 1 \ldots n}]}, \quad \nu_i \in \{^+, ^-, ^0\}, B_i$ covariant in X

(Sub Object) For $A \equiv Obj(X)[l_i \nu_i : B_i\{X\} \ ^{i \in 1 \ldots n+m}], A' \equiv Obj(X)[l_i \nu'_i : B'_i\{X\} \ ^{i \in 1 \ldots n}]$

$$\frac{E, Y <: A \vdash \nu_i B_i\{Y\} <: \nu'_i B'_i\{Y\} \quad \forall i \in 1 \ldots n}{E \vdash A <: A'}$$

Term Typings

(Val Object) $\dfrac{E, x_i : A \vdash b_i\{A\} : B_i\{A\} \quad \forall i \in 1 \ldots n}{E \vdash obj(X = A)[l_i = \varsigma(x_i : X) b_i\{X\} \ ^{i \in 1 \ldots n}] : A}, \quad A \equiv Obj(X)[l_i \nu_i : B_i\{X\} \ ^{i \in 1 \ldots n}]$

(Struct Val Select) For $A' \equiv Obj(X)[l_i \nu_i : B_i\{X\} \ ^{i \in 1 \ldots n}], \nu_j \in \{^+, ^0\}, j \in 1 \ldots n$

$$\frac{E \vdash a : A \quad E \vdash A <: A'}{E \vdash a.l_j : B_j\{A\}}$$

(Struct Val Update) For $A' \equiv Obj(X)[l_i \nu_i : B_i\{X\} \ ^{i \in 1 \ldots n}], \nu_j \in \{^-, ^0\}, j \in 1 \ldots n$

$$\frac{E \vdash a : A \quad E \vdash A <: A' \quad E, Y <: A, y : Y, x : Y \vdash b : B_j\{Y\}}{E \vdash a.l_j \Leftarrow (Y <: A, y : Y) \varsigma(x : Y) b : A}$$