# Part III

# FASE

# An Imperative Object Calculus

Martín Abadi and Luca Cardelli

Digital Equipment Corporation, Systems Research Center

**Abstract.** We develop an imperative calculus of objects. Its main type constructor is the one for object types, which incorporate variance annotations and Self types. A subtyping relation between object types supports object subsumption. The type system for objects relies on unusual but beneficial assumptions about the possible subtypes of an object type. With the addition of polymorphism, the calculus can express classes and inheritance.

## 1. Introduction

Object calculi are formalisms at the same level of abstraction as λ-calculi, but based exclusively on objects rather than functions. Unlike λ-calculi, object calculi are designed specifically for clarifying features of object-oriented languages. There is a wide spectrum of relevant object calculi, just as there is a wide spectrum of λ-calculi. One can investigate untyped, simply-typed, and polymorphic calculi, as well as functional and imperative calculi.

In object calculi, as in λ-calculi, a minimal untyped kernel is enriched with derived constructions and with increasingly sophisticated type systems, until language features can be realistically modeled. The compactness of the initial kernel gives conceptual unity to the calculi, and enables formal analysis.

In this paper, we develop an imperative object calculus. It provides a minimal setting in which to study the imperative operational semantics and the delicate typing rules of practical object-oriented languages.

The calculus includes objects, method invocation, method update, and object cloning. Traditionally, an object is seen as a bundle of mutable fields together with a method suite. In our calculus, the method suite is itself mutable, so we can dispense with the fields.

The main type constructor is the one for object types; an object type is a list of method names and method result types. A subtyping relation between object types supports object subsumption, which allows an object to be used where an object with fewer methods is expected. Variance annotations enable flexible subtyping and protection from side effects.

The object type constructor incorporates a notion of Self types. Intuitively, Self is the partially unknown type of the self parameter of each method. Several object-oriented languages have included it in their type systems [18, 22], sometimes with unsound rules [13]. Therefore, it seems important to have a precise understanding of Self. Unfortunately, it has proven hard to reduce Self to more primitive and well-understood notions (see [3, 7, 19] for recent progress). We aim to provide a satisfactory treatment of Self by taking it as primitive and axiomatizing its desired properties.

The treatment of Self types relies on assumptions about the possible subtypes of object types. These assumptions are operationally sound, but would not hold in natural semantic models. We show the necessity of these assumptions in finding satisfactory typings for programs involving Self types.

We consider also bounded type quantifiers for polymorphism. Taken together, objects with imperative features, object types, and polymorphism form a realistic kernel for a programming language. Using these primitives, we account for classes, subclasses, inheritance, method and field specialization in subclasses, parametric method update, and protection from external updates.

A few other object formalisms have been defined and studied. Many of these rely on purely functional models, with an emphasis on types [1, 7, 10, 11, 16, 19-21, 24]. Others deal with imperative features in the context of concurrency; see for example [26]. The works most closely related to ours are that of Eifrig *et al.* on LOOP [14] and that of Bruce *et al.* on PolyTOIL [9]. LOOP and PolyTOIL are typed, imperative, object-oriented languages with procedures, objects, and classes. PolyTOIL takes procedures, objects, and classes as primitive, with fairly complex rules. LOOP is translated into a somewhat simpler calculus. Our calculus is centered on objects; procedures and classes can be defined from them. Despite these differences, we all share the goal of modeling imperative object-oriented languages by precise semantic structures and sound type systems.

This paper is self-contained, but continues our work of [2-5]. The most apparent novelties are the variance annotations, the treatment of Self types, and the representation of classes and inheritance. The new typing features led us to prefer syntactic proof techniques over denotational methods.

In section 2 we give the untyped term syntax of our calculus and its operational semantics. In section 3 we present object types. In section 4 we add bounded universal quantifiers. In section 5 we discuss soundness. In section 6 we consider the typing of some critical examples, and provide a representation of classes and method inheritance.

## 2. An Untyped Imperative Calculus

We begin with the syntax of an untyped imperative calculus. The initial syntax is minimal, but in sections 2.2, 2.3, and 6.2 we show how to express convenient constructs such as fields, procedures, and classes. We omit how to encode basic data types and control structures, which can be treated much as in [4]. In section 2.5 we give an operational semantics.

### 2.1 Syntax and Informal Semantics

The evaluation of terms is based on an imperative operational semantics with a store, and generally proceeds deterministically from left to right. The letter $\varsigma$ (sigma) is a binder; it delays evaluation of the term to its right.

***Syntax of terms***

| | |
|---|---|
| $a,b ::=$ | term |
| $x$ | variable |
| $[l_i=\varsigma(x_i)b_i\ ^{i\in 1\ n}]$ | object ($l_i$ distinct) |
| $a.l$ | method invocation |
| $a.l \Leftarrow (y,\ z=c)\varsigma(x)b$ | method update |
| $clone(a)$ | cloning |

An object is a collection of components $l_i=\varsigma(x_i)b_i$, for distinct labels $l_i$ and associated methods $\varsigma(x_i)b_i$; the order of these components does not matter, even for our deterministic operational semantics. Each binder $\varsigma$ binds the self parameter of a method; $\varsigma(x)b$ is a method with self variable x and body b.

A method invocation b.l results in the evaluation of b, followed by the evaluation of the body of the method named l, with the value of b bound to the self variable of the method.

A cloning operation clone(b) produces a new object with the same labels as b, with each component sharing the methods of the corresponding component of b.

The method update construct, $a.l \Leftarrow (y,z{=}c)\varsigma(x)b$, is best understood by looking at two of its special cases:

***Simple method update***: $a.l \Leftarrow \varsigma(x)b$

This construct evaluates a, replaces the method named l with the new method $\varsigma(x)b$, and returns the modified object.

***Method update with old self***: $a.l \Leftarrow (y)\varsigma(x)b$

This construct is similar to simple method update, but in addition the value of a is bound to the variable y within $\varsigma(x)b$. Informally, it means $y.l \Leftarrow \varsigma(x)b$ where the value of a is bound to y. After the update, when the method l is invoked, y still points to the old value of a, and x points to the current self.

The general method update construct, $a.l \Leftarrow (y,z{=}c)\varsigma(x)b$, first evaluates a and binds its value to y, then evaluates c and binds its value to z, and finally updates the l method of y and returns the modified object. The variable y may occur in c and b, and the variables z and x may occur in b.

The two special cases above are definable from $a.l \Leftarrow (y,z{=}c)\varsigma(x)b$:

$$a.l \Leftarrow \varsigma(x)b \triangleq a.l \Leftarrow (y, z{=}y)\varsigma(x)b \qquad \text{where } y,z \notin FV(b)$$

$$a.l \Leftarrow (y)\varsigma(x)b \triangleq a.l \Leftarrow (y, z{=}y)\varsigma(x)b \qquad \text{where } z \notin FV(b)$$

The standard let and sequencing constructs are also definable:

$$\text{let } x = a \text{ in } b \triangleq$$

$$([val = \varsigma(y)y.val].val \Leftarrow (z, x{=}a)\varsigma(w)b).val \qquad \text{where } z,w \notin FV(b) \text{ and } z \notin FV(a)$$

$$a\, ;\, b \triangleq \text{let } x = a \text{ in } b \qquad \text{where } x \notin FV(b)$$

Conversely, in an untyped calculus, the construct $a.l \Leftarrow (y,z{=}c)\varsigma(x)b$ can be expressed in terms of let and simple method update, as let y = a in let z = c in $y.l \Leftarrow \varsigma(x)b$. However, the construct $a.l \Leftarrow (y,z{=}c)\varsigma(x)b$ yields better typings, as shown in section 6.1.

## 2.2 Fields

In our calculus, every component of an object contains a method. However, we can encode fields with eagerly evaluated contents. We write $[l_i{=}b_i\ ^{i\in 1..n}, l_j{=}\varsigma(x_j)b_j\ ^{j\in 1..m}]$ for an object where $l_i{=}b_i$ are fields and $l_j{=}\varsigma(x_j)b_j$ are methods. We also write a.l:=b for field update, and a.l, as before, for field selection. We abbreviate:

***Field notation***

$$[l_i{=}b_i\ ^{i\in 1..n}, l_j{=}\varsigma(x_j)b_j\ ^{j\in 1..m}] \qquad \text{for } y_i \notin FV(b_i\ ^{i\in 1..n}, b_j\ ^{j\in 1..m}),\ y_i \text{ distinct, } i\in 0..n$$

$$\triangleq \text{ let } y_1{=}b_1 \text{ in} \ldots \text{let } y_n{=}b_n \text{ in } [l_i{=}\varsigma(y_0)y_i\ ^{i\in 1..n}, l_j{=}\varsigma(x_j)b_j\ ^{j\in 1..m}]$$

$$a.l{:=}b \quad \triangleq \quad a.l \Leftarrow (y, z{=}b)\varsigma(x)z \qquad \text{for } y \notin FV(b),\ x,y,z \text{ distinct}$$

The semantics of an object with fields may depend on the order of its components, because of side-effects in computing contents of fields. The encoding specifies an evaluation order.

By an update, a method can be changed into a field and vice versa. Thus, we use somewhat interchangeably the names selection and invocation.

## 2.3 Procedures

Our object calculus is so minimal that it does not include procedures, but these can be expressed too. To illustrate this point, we consider informally an imperative call-by-value λ-calculus that includes abstraction, application, and assignment to λ-bound variables. For example, assuming arithmetic primitives, (λ(x) x:=x+1)(3) is a term yielding 4. We translate this λ-calculus into our object calculus:

***Translation of procedures***

$$\begin{aligned}
\langle\!\langle x\rangle\!\rangle_\rho &\triangleq \rho(x) \text{ if } x \in dom(\rho), \text{ and } x \text{ otherwise}\\
\langle\!\langle x{:=}a\rangle\!\rangle_\rho &\triangleq x.arg{:=}\langle\!\langle a\rangle\!\rangle_\rho\\
\langle\!\langle \lambda(x)b\rangle\!\rangle_\rho &\triangleq [arg = \varsigma(x)x.arg,\ val = \varsigma(x)\langle\!\langle b\rangle\!\rangle_{\rho\{x\leftarrow x.arg\}}]\\
\langle\!\langle b(a)\rangle\!\rangle_\rho &\triangleq (clone(\langle\!\langle b\rangle\!\rangle_\rho).arg{:=}\langle\!\langle a\rangle\!\rangle_\rho).val
\end{aligned}$$

In the translation, an environment ρ maps each variable x either to x.arg if x is λ-bound, or to x if x is a free variable. A λ-abstraction is translated to an object with an arg component, for storing the argument, and a val method, for executing the body. The arg component is initially set to a divergent method, and is filled with an argument upon procedure call. A call activates the val method that can then access the argument through self as x.arg. An assignment x:=a updates x.arg, where the argument is stored (assuming that x is λ-bound). A procedure needs to be cloned when it is called; the clone provides a fresh location in which to store the argument of the call, preventing interference with other calls of the same procedure. Such interference would derail recursive invocations.

## 2.4 A Small Example

We give a trivial example as a notation drill. We use fields, procedures, and booleans in defining a memory cell with get, set, and dup (duplicate) components:

let m = [get = false, set = ς(self) λ(b) self.get:=b, dup = ς(self) clone(self)]
in m.set(true); m.get yields true

## 2.5 Operational Semantics

We now give an operational semantics for the basic calculus of section 2.1; the semantics relates terms to results in a global store. Object terms reduce to object results $[l_i{=}\iota_i{}^{i\in 1..n}]$ consisting of sequences of store locations, one location for each object component. In order to stay close to standard implementation techniques, we avoid using formal substitutions during reduction: we describe a semantics based on stacks and closures. A stack $\mathcal{S}$ associates variables with results; a closure $\langle\varsigma(x)b,S\rangle$ is a pair of a method and a stack that is used for the reduction of the method body. A store maps locations $\iota$ to method closures; we write stores in the form $\iota_i\mapsto\langle\varsigma(x_i)b_i,S_i\rangle^{\,i\in 1..n}$. We let $\sigma.\iota\leftarrow m$ denote the result of writing m in the $\iota$ location of $\sigma$.

The operational semantics is expressed in terms of a relation that relates a store $\sigma$, a stack $S$, a term b, a result v, and another store $\sigma'$. This relation is written $\sigma\cdot S\vdash b \leadsto v\cdot\sigma'$, and it means that with the store $\sigma$ and the stack $S$, the term b reduces to a result v, yielding an updated store $\sigma'$; the stack does not change.

*Notation*

| | | | | |
|---|---|---|---|---|
| $\iota$ | | | store location | (e.g., an integer) |
| $v$ | $::=$ | $[l_i=\iota_i\ ^{i\in 1..n}]$ | object result | ($l_i$ distinct) |
| $\sigma$ | $::=$ | $\iota_i\mapsto\langle\varsigma(x_i)b_i,S_i\rangle^{\,i\in 1..n}$ | store | ($\iota_i$ distinct) |
| $S$ | $::=$ | $x_i\mapsto v_i\ ^{i\in 1..n}$ | stack | ($x_i$ distinct) |

***Well-formed store judgment:*** $\sigma\vdash\diamond$

(Store $\emptyset$)

$$\frac{}{\emptyset\vdash\diamond}$$

(Store $\iota$)

$$\frac{\sigma\cdot S\vdash\diamond \qquad \iota\notin dom(\sigma)}{\sigma,\ \iota\mapsto\langle\varsigma(x)b,S\rangle\vdash\diamond}$$

***Well-formed stack judgment:*** $\sigma\cdot S\vdash\diamond$

(Stack $\emptyset$)

$$\frac{\sigma\vdash\diamond}{\sigma\cdot\emptyset\vdash\diamond}$$

(Stack x) ($l_i,\iota_i$ distinct)

$$\frac{\sigma\cdot S\vdash\diamond \qquad x\notin dom(S) \qquad \iota_i\in dom(\sigma) \qquad \forall i\in 1..n}{\sigma\cdot S,\ x\mapsto[l_i=\iota_i\ ^{i\in 1..n}]\vdash\diamond}$$

***Term reduction judgment:*** $\sigma\cdot S\vdash a\leadsto v\cdot\sigma'$

(Red x)

$$\frac{\sigma\cdot S',x\mapsto v,S''\vdash\diamond}{\sigma\cdot S',x\mapsto v,S''\vdash x\leadsto v\cdot\sigma}$$

(Red Object) ($l_i,\iota_i$ distinct)

$$\frac{\sigma\cdot S\vdash\diamond \qquad \iota_i\notin dom(\sigma) \qquad \forall i\in 1..n}{\sigma\cdot S\vdash[l_i=\varsigma(x_i)b_i\ ^{i\in 1..n}]\leadsto[l_i=\iota_i\ ^{i\in 1..n}]\cdot(\sigma,\ \iota_i\mapsto\langle\varsigma(x_i)b_i,S\rangle^{\,i\in 1..n})}$$

(Red Select)

$$\frac{\begin{array}{c}\sigma\cdot S\vdash a\leadsto[l_i=\iota_i\ ^{i\in 1..n}]\cdot\sigma' \qquad \sigma'(\iota_j)=\langle\varsigma(x_j)b_j,S'\rangle \qquad x_j\notin dom(S') \qquad j\in 1..n\\ \sigma'\cdot S',\ x_j\mapsto[l_i=\iota_i\ ^{i\in 1..n}]\vdash b_j\leadsto v\cdot\sigma''\end{array}}{\sigma\cdot S\vdash a.l_j\leadsto v\cdot\sigma''}$$

(Red Update)

$$\frac{\sigma{\bullet}S \vdash a \rightsquigarrow [l_i{=}\iota_i\ ^{i\in 1..n}]{\bullet}\sigma' \quad \iota_j{\in}\mathrm{dom}(\sigma') \quad j{\in}1..n \qquad \sigma'{\bullet}S,\ y{\mapsto}[l_i{=}\iota_i\ ^{i\in 1..n}] \vdash c \rightsquigarrow v{\bullet}\sigma''}{\sigma{\bullet}S \vdash a.l_j{\Leftarrow}(y,\ z{=}c)\varsigma(x)b \rightsquigarrow [l_i{=}\iota_i\ ^{i\in 1..n}] \bullet \sigma''.\iota_j{\leftarrow}\langle\varsigma(x)b,\ (S,\ y{\mapsto}[l_i{=}\iota_i\ ^{i\in 1..n}],\ z{\mapsto}v)\rangle}$$

(Red Clone) ($\iota'_i$ distinct)

$$\frac{\sigma{\bullet}S \vdash a \rightsquigarrow [l_i{=}\iota_i\ ^{i\in 1..n}]{\bullet}\sigma' \quad \iota_i \in \mathrm{dom}(\sigma') \quad \iota'_i \notin \mathrm{dom}(\sigma) \quad \forall i{\in}1..n}{\sigma{\bullet}S \vdash \mathrm{clone}(a) \rightsquigarrow [l_i{=}\iota'_i\ ^{i\in 1..n}] \bullet (\sigma',\ \iota'_i{\mapsto}\sigma'(\iota_i)\ ^{i\in 1..n})}$$

A variable reduces to the result it denotes in the current stack. An object reduces to a result consisting of a fresh collection of locations; the store is extended to associate method closures to those locations. A selection operation $a.l_j$ first reduces the object a to a result, then activates the appropriate method closure. An update operation $a.l_j{\Leftarrow}(y,z{=}c)\varsigma(x)b$ first reduces the object a to the final result; next, with y bound to that result, it reduces the term c to another result; finally, it updates the appropriate store location with a closure consisting of the new method $\varsigma(x)b$ and a stack binding y and z. A clone operation reduces its object to a result; then it allocates a fresh collection of locations that are associated to the existing method closures from the object.

We illustrate method update, and the creation of loops through the store, by the reduction of the term $[l{=}\varsigma(x)x.l{:=}x].l$, that is, $[l{=}\varsigma(x)x.l{\Leftarrow}(y,z{=}x)\varsigma(w)z].l$:

$$\emptyset{\bullet}\emptyset \vdash [l{=}\varsigma(x)x.l{\Leftarrow}(y,z{=}x)\varsigma(w)z].l \rightsquigarrow [l{=}0]{\bullet}\sigma$$
$$\text{where} \quad \sigma \equiv 0{\mapsto}\langle\varsigma(w)z,\ (x{\mapsto}[l{=}0],\ y{\mapsto}[l{=}0],\ z{\mapsto}[l{=}0])\rangle$$

The store $\sigma$ contains a loop, because it maps the index 0 to a closure that binds the variable z to a value that contains index 0. Hence, an attempt to read out the result of $[l{=}\varsigma(x)x.l{:=}x].l$ by "inlining" the store and stack mappings would produce the infinite term $[l{=}\varsigma(w)[l{=}\varsigma(w)[l{=}\varsigma(w)...]]]$.

## 3. Typing

In this section we develop a type system for the calculus of section 2. We have the following syntax of types:

***Syntax of types***

| | |
|---|---|
| A,B ::= | type |
| X | type variable |
| Top | the biggest type |
| $\mathrm{Obj}(X)[l_i\upsilon_i{:}B_i\{X\}\ ^{i\in 1..n}]$ | object type ($\upsilon_i \in \{^-,^\circ,^+\}$, $l_i$ distinct) |

Let $A \equiv \mathrm{Obj}(X)[l_i\upsilon_i{:}B_i\{X\}\ ^{i\in 1..n}]$. The binder Obj binds a Self type named X, which is known to be a subtype of A. Then A is the type of those objects with methods named $l_i\ ^{i\in 1..n}$ having self parameters of type X, and with corresponding result types $B_i\{X\}$. The type X may occur only covariantly in the result types $B_i$. This covariance requirement is necessary for the soundness of our rules; covariance is defined precisely below.

Each $\upsilon_i$ is a variance annotation; it is one of the symbols $^-$, $^o$, and $^+$, for contravariance, invariance, and covariance, respectively. Covariant components allow covariant subtyping, but prevent update. Symmetrically, contravariant components allow contravariant subtyping, but prevent invocation. Invariant components can be both invoked and updated; by subtyping, they can be regarded as either covariant or contravariant. Therefore, variance annotations support flexible subtyping and a form of protection.

The rules for our object calculus are given next. The first three groups of rules concern typing environments, types, and the subtyping relation. The final group concerns typing of terms: there is one rule for each construct in the calculus; in addition, a subsumption rule connects term typing with subtyping.

***Well-formed environment judgment:*** $E \vdash \diamond$

(Env $\emptyset$)
$$\frac{}{\emptyset \vdash \diamond}$$

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

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

***Well-formed type judgment:*** $E \vdash A$

(Type X<:)
$$\frac{E', X{<:}A, E'' \vdash \diamond}{E', X{<:}A, E'' \vdash X}$$

(Type Top)
$$\frac{E \vdash \diamond}{E \vdash Top}$$

(Type Object) ($l_i$ distinct, $\upsilon_i \in \{^o, ^-, ^+\}$)
$$\frac{E, X{<:}Top \vdash B_i\{X^+\} \qquad \forall i \in 1..n}{E \vdash Obj(X)[l_i \upsilon_i{:}B_i\{X\}\ ^{i\in 1..n}]}$$

Formally, $B\{X^+\}$ indicates that X occurs only covariantly in B; that is, either B is a variable (possibly X), or $B \equiv Top$, or $B \equiv Obj(Y)[l_i \upsilon_i{:}B_i\ ^{i\in 1..n}]$ and either $Y \equiv X$ or for each $\upsilon_i \equiv ^+$ we have $B_i\{X^+\}$, for each $\upsilon_i \equiv ^-$ we have $B_i\{X^-\}$, and for each $\upsilon_i \equiv ^o$ we have $X \notin FV(B_i)$. Similarly, $B\{X^-\}$ indicates that X occurs only contravariantly in B; that is, either B is a variable different from X, or $B \equiv Top$, or $B \equiv Obj(Y)[l_i \upsilon_i{:}B_i\ ^{i\in 1..n}]$ and either $Y \equiv X$ or for each $\upsilon_i \equiv ^+$ we have $B_i\{X^-\}$, for each $\upsilon_i \equiv ^-$ we have $B_i\{X^+\}$, and for each $\upsilon_i \equiv ^o$ we have $X \notin FV(B_i)$.

The formation rule for object types (Type Object) requires that all the component types be covariant in Self. According to the definition of covariant occurrences, more than one Self type may be active in a given context, as in the type $Obj(X)[l^o{:}\ Obj(Y)[m^+{:}X, n^o{:}Y]]$.

By convention, any omitted $\upsilon$'s are taken to be equal to $^o$. We regard a simple object type $[l_i{:}B_i\ ^{i\in 1..n}]$ (as defined in [4]) as an abbreviation for $Obj(X)[l_i^o{:}B_i\ ^{i\in 1..n}]$, where X does not appear in any $B_i$.

***Subtyping judgments:*** $E \vdash A <: B$, $E \vdash \upsilon A <: \upsilon B$

(Sub Refl)
$$\frac{E \vdash A}{E \vdash A <: A}$$

(Sub Trans)
$$\frac{E \vdash A <: B \qquad E \vdash B <: C}{E \vdash A <: C}$$

(Sub X)
$$\frac{E', X{<:}A, E'' \vdash \diamond}{E', X{<:}A, E'' \vdash X{<:}A}$$

(Sub Top)
$$\frac{E \vdash A}{E \vdash A <: Top}$$

(Sub Object)

$$\frac{E, Y <: Obj(X)[l_i \upsilon_i : B_i\{X\}^{i \in 1..n+m}] \vdash \upsilon_i B_i\{Y\} <: \upsilon_i' B_i'\{Y\} \qquad \forall i \in 1..n}{E \vdash Obj(X)[l_i \upsilon_i : B_i\{X\}^{i \in 1..n+m}] <: Obj(X)[l_i \upsilon_i' : B_i'\{X\}^{i \in 1..n}]}$$

(Sub Invariant)

$$\frac{E \vdash B}{E \vdash {}^{o}B <: {}^{o}B}$$

(Sub Covariant)

$$\frac{E \vdash B <: B' \qquad \upsilon \in \{{}^{o},{}^{+}\}}{E \vdash \upsilon B <: {}^{+}B'}$$

(Sub Contravariant)

$$\frac{E \vdash B' <: B \qquad \upsilon \in \{{}^{o},{}^{-}\}}{E \vdash \upsilon B <: {}^{-}B'}$$

The subtyping rule for object types (Sub Object) says, to a first approximation, that a longer object type on the left is a subtype of a shorter one on the right. The antecedents operate under the assumption that Self is a subtype of the longer type.

Because of the variance annotations we use an auxiliary judgment, $E \vdash \upsilon B <: \upsilon' B'$, for inclusion of components with variance. The rules say:

- (Sub Object) Components that occur both on the left and on the right are handled by the other three rules. For components that occur only on the left, the component types must be well-formed.
- (Sub Invariant) An invariant component on the right requires an identical one on the left.
- (Sub Covariant) A covariant component type on the right can be a supertype of a corresponding component type on the left, either covariant or invariant. Intuitively, an invariant component can be regarded as covariant.
- (Sub Contravariant) A contravariant component type on the right can be a subtype of a corresponding component type on the left, either contravariant or invariant. Intuitively, an invariant component can be regarded as contravariant.

The type Obj(X)[...] can be viewed as a recursive type, but with differences in subtyping that are crucial for object-oriented applications. The subtyping rule for object types (Sub Object), with all components invariant, would read:

$$\frac{E, X <: Top \vdash B_i\{X^+\} \qquad \forall i \in 1..n+m}{E \vdash Obj(X)[l_i : B_i\{X\}^{i \in 1..n+m}] <: Obj(X)[l_i : B_i\{X\}^{i \in 1..n}]}$$

An analogous rule would be unsound with recursive types instead of Self types [6].

***Term typing judgment:*** $E \vdash a : A$

(Val Subsumption)

$$\frac{E \vdash a : A \qquad E \vdash A <: B}{E \vdash a : B}$$

(Val x)

$$\frac{E', x:A, E'' \vdash \diamond}{E', x:A, E'' \vdash x:A}$$

(Val Object) (where $A \equiv Obj(X)[l_i \upsilon_i : B_i\{X\}^{i \in 1..n}]$)

$$\frac{E, x_i:A \vdash b_i : B_i\{A\} \qquad \forall i \in 1..n}{E \vdash [l_i = \varsigma(x_i)b_i^{\,i \in 1..n}] : A}$$

(Val Select) (where $A' \equiv Obj(X)[l_i \upsilon_i : B_i\{X\}^{i \in 1..n}]$)

$$\frac{E \vdash a : A \qquad E \vdash A <: A' \qquad \upsilon_j \in \{^\circ, ^+\} \qquad j \in 1..n}{E \vdash a.l_j : B_j\{A\}}$$

(Val Update) (where $A' \equiv Obj(X)[l_i \upsilon_i : B_i\{X\}^{i \in 1..n}]$)

$$\frac{\begin{array}{c} E \vdash a : A \qquad E \vdash A <: A' \qquad E, Y <: A, y : Y \vdash c : C \\ E, Y <: A, y : Y, z : C, x : Y \vdash b : B_j\{Y\} \qquad \upsilon_j \in \{^\circ, ^-\} \qquad j \in 1..n \end{array}}{E \vdash a.l_j \Leftarrow (y, z = c)\varsigma(x)b : A}$$

(Val Clone) (where $A' \equiv Obj(X)[l_i \upsilon_i : B_i\{X\}^{i \in 1..n}]$)

$$\frac{E \vdash a : A \qquad E \vdash A <: A'}{E \vdash clone(a) : A}$$

The typing rules are largely the same ones we would have for a functional calculus; the main novelty is the construct for method update. Because of the eager evaluation associated with imperative semantics, there is a need for a construct that can express sequential evaluation. This construct is combined with method update to obtain sufficiently general typings.

To preserve soundness, the rules for selection and update are restricted: selection cannot operate on contravariant components, while update cannot operate on covariant components.

A remarkable aspect of our type system is that the rules (Val Select), (Val Update), and (Val Clone) are based on structural assumptions about the universe of types (cf. [3]). These assumptions are operationally valid, but would not hold in natural semantic models. For example, the update rule implies that if x:X and X<:A' where A' is a given object type with an invariant component l:B, then we may update l in x with a term b of type B yielding an updated object of type X, and not just A'. This rule is based on the assumption that any X<:A' is closed under updating of l with elements of B. The closure property holds in a model only if any subtype of A' allows the result of l to be any element of B. Intuitively, this condition may fail because a subtype of A' may be a subset with l:B' for B' strictly included in B. Operationally, the closure property holds because any possible instance of X in the course of a computation is a closed object type that, being a subtype of A', has a component l of type exactly B.

Rules based on structural assumptions (structural rules, for short) are critical for Self types; they are required for typing programs satisfactorily. Structural rules allow methods to act parametrically over any X<:A', where X is the Self type and A' is a given object type. In section 6 we demonstrate the power of structural rules by examples, and by our representation of classes and inheritance. We can prove that structural rules are sound for our operational semantics (see section 5).

## 4. Self and Polymorphism

Self types produce an expressive type system, sufficient for interesting examples. However, this type system still lacks facilities for type parameterization. Type parameterization has useful interactions with objects; in particular, it supports method reuse and inheritance, as we show in section 6. To allow for parameterization, we add bounded universal quantifiers [12].

First we extend the syntax of terms. We add two new forms. We write λ()b for a type abstraction and a() for a type application. In typed calculi, it is common to find λ(X<:A)b and a(A) instead. However, we are already committed to an untyped operational semantics, so we strip the types from those terms. Technically, we adopt λ()b, instead of dropping λ() altogether, in order to distinguish the elements of quantified types from those of object types. The distinction greatly simplifies case analysis in proofs.

***Additional syntax of terms***

| | | |
|---|---|---|
| a,b ::= | term | |
| . . . | | (as before) |
| λ()b | | type abstraction |
| a() | | type application |

This choice of syntax affects the operational semantics. In particular, a closure ⟨λ()b,$\mathcal{S}$⟩, consisting of a type abstraction and a stack, is a result:

***Additional results***

| | | |
|---|---|---|
| v ::= | result | |
| . . . | | (as before) |
| ⟨λ()b,$\mathcal{S}$⟩ | | type abstraction result |

We add two rules to the operational semantics. According to these rules, evaluation stops at type abstractions and is triggered again by type applications. This is a sensible semantics of polymorphism, particularly in presence of side-effects.

***Additional term reductions***

(Red Fun2)

$$\frac{\sigma{\bullet}\mathcal{S} \vdash \diamond}{\sigma{\bullet}\mathcal{S} \vdash \lambda()b \rightsquigarrow \langle\lambda()b,\mathcal{S}\rangle{\bullet}\sigma}$$

(Red Appl2)

$$\frac{\sigma{\bullet}\mathcal{S} \vdash a \rightsquigarrow \langle\lambda()b,\mathcal{S}'\rangle{\bullet}\sigma' \qquad \sigma'{\bullet}\mathcal{S}' \vdash b \rightsquigarrow v{\bullet}\sigma''}{\sigma{\bullet}\mathcal{S} \vdash a() \rightsquigarrow v{\bullet}\sigma''}$$

The typing rules for bounded universal quantifiers are:

***Additional typing rules***

(Type All<:)

$$\frac{E,X{<:}A \vdash B}{E \vdash \forall(X{<:}A)B}$$

(Sub All)

$$\frac{E \vdash A' <: A \qquad E,X{<:}A' \vdash B <: B'}{E \vdash \forall(X{<:}A)B <: \forall(X{<:}A')B'}$$

(Val Fun2<:)

$$\frac{E,X{<:}A \vdash b : B}{E \vdash \lambda()b : \forall(X{<:}A)B}$$

(Val Appl2<:)

$$\frac{E \vdash b : \forall(X{<:}A)B\{X\} \qquad E \vdash A'{<:}A}{E \vdash b() : B\{A'\}}$$

The variance of quantifiers, implied by (Sub All), is the usual one: $\forall(X<:A)B$ is contravariant in the bound (A) and covariant in the body (B).

## 5. Soundness

Our technique for proving typing soundness is an extension of Harper's [15], but using closures and stacks instead of formal substitutions (see [17, 23, 25] for related techniques). This approach yields a manageable proof for a realistic implementation strategy, and deals easily with typing rules that seem hard to justify denotationally. Our soundness result covers subtyping and polymorphism in the presence of side-effects.

Here we present only a simple statement of a subject reduction theorem: additional definitions are needed for a proper formulation of the induction hypothesis.

**Subject Reduction Theorem**

If $\emptyset \vdash a : \mathrm{Obj}(X)[l_i \upsilon_i : B_i\ ^{i \in 1..n}]$ and $\emptyset \bullet \emptyset \vdash a \leadsto v \bullet \sigma$ then v is an object result.
If $\emptyset \vdash a : \forall(X<:A)B$ and $\emptyset \bullet \emptyset \vdash a \leadsto v \bullet \sigma$ then v is a type abstraction result.

The proof is an extension of the one given in [5] for an imperative calculus with a simpler type structure. It is based on store types [23], and on judgments for typing a result with respect to a store type, and for typing a stack of closed types and a stack of results with respect to a typing environment.

## 6. Applications

We study some challenging examples. Then we show that quantifiers are useful for factoring out methods as generic procedures, and we describe how collections of generic procedures can be organized into classes and subclasses.

### 6.1 Typing Challenges and Solutions

We examine some delicate typing issues in the context of simple examples. We use procedures, as defined in section 2.3, and booleans. Procedure types can be defined as:

$$A \to B \triangleq [arg^{-}:A, val^{+}:B]$$

The variance annotations yield the expected contravariant/covariant subtyping rule for procedure types. The typing of procedures relies on the inclusion $[arg^{\circ}: A, val^{\circ}: B] <: A \to B$.

• We define the type of memory cells with a get field and with a set method. Given a boolean, the set method stores it in the cell and returns the modified cell. For get, we use a method. (Using a field, with the encoding of section 2.2, leads to simpler code but makes it harder to check the typings of the examples against the rules.)

$$\mathrm{Mem} \triangleq \mathrm{Obj}(X)[get:Bool, set:Bool \to X]$$
$$m: \mathrm{Mem} \triangleq [get = \varsigma(x)\ false, set = \varsigma(x)\ \lambda(b)\ x.get \Leftarrow \varsigma(z)\ b]$$

• Interesting uses of (Val Update) are required when updating methods that return a value of type Self. Here we update the method set with a method that updates get to $\varsigma(z)$false:

$m.set \Leftarrow \varsigma(x)\, \lambda(b)\, x.get \Leftarrow \varsigma(z)\, false \quad : \quad Mem$

To obtain this typing using (Val Update), we give type Y to x.get⇐ς(z)false for an arbitrary Y<:Mem, parametrically in Y.

• With quantifiers, we can define "parametric pre-methods" as polymorphic procedures that can later be used in updating. The method previously used for updating m.set, for example, can be isolated as a procedure of type ∀(X<:Mem)X→X:

$\lambda()\, \lambda(m)\, m.set \Leftarrow \varsigma(x)\, \lambda(b)\, x.get \Leftarrow \varsigma(z)\, false \quad : \quad \forall(X<:Mem)X\to X$

The derivation makes an essential use of the structural subtyping assumption in (Val Update); without it we would obtain at best the type ∀(X<:Mem)X→Mem.

• It is natural to expect both components of Mem to be protected against external update. To this end we can use covariance annotations, which block (Val Update). Take:

$ProtectedMem \quad \triangleq \quad Obj(X)[get^{+}:Bool,\ set^{+}:Bool\to X]$

Since Mem <: ProtectedMem, any memory cell can be subsumed into ProtectedMem and thus protected against updating from the outside. However, after subsumption, the set method can still update the get field because it was originally typechecked with X equal to Mem.

• Consider a type of memory cells with a duplicate method, as in section 2.4.

$MemDup \quad \triangleq \quad Obj(X)[get:Bool,\ set:Bool\to X,\ dup:X]$

We have MemDup <: Mem, thanks to the subtyping rule for object types. This subtyping would have failed had we used recursive type instead of Self types [4].

• Consider now a type of memory cells with backup and restore methods, and a candidate implementation:

$MemBk \quad \triangleq \quad Obj(X)[restore:X,\ backup:X,\ get:Bool,\ set:Bool\to X]$

$o \quad \triangleq \quad [restore = \varsigma(self)\, self,\ backup = \varsigma(self)\, self.restore := clone(self),$
$\qquad get = \ldots,\ set = \ldots\ ]$

The initial restore method is set to return the current memory cell. Whenever the backup method is invoked, it places a clone of self into restore. Note that backup saves the self that is current at backup-invocation time, not the self that will be current at restore-invocation time.

The untyped behavior of backup and restore are the desired ones, but o cannot be given the type MemBk. We can see why by expanding the definition of :=, obtaining:

$backup = \ \varsigma(self)\, self.restore \Leftarrow (y,\ z{=}clone(self))\ \varsigma(x)\, z$

In typing self.restore⇐(y,z=clone(self))ς(x)z, we have self:MemBk and z:MemBk as well. The update requires that for an arbitrary Y<:MemBk we be able to show that z:Y. This cannot be achieved.

The following alternative code has the same problem if we assume the obvious typing rule for let:

backup $=$ ς(self) let z = clone(self) in self.restore $\Leftarrow$ ς(x) z

Therefore, for typing this example, it is not sufficient to adopt let and simple update as separate primitives.

The solution to this typing problem requires the general method update construct:

backup = ς(self) self.restore $\Leftarrow$ (y, z=clone(y)) ς(x) z

In typing self.restore$\Leftarrow$(y,z=clone(y))ς(x)z, we have self:MemBk. For an arbitrary Y<:MemBk, the update rule assigns to y the type Y. Therefore, clone(y) has type Y by (Val Clone), and hence z has the required type Y.

Note that this typing problem manifests itself with field update, and is not a consequence of allowing method update. It arises from the combination of Self and eager evaluation, when a component of a type that depends on Self is updated.

## 6.2 Classes as Collections of Pre-Methods

As shown in section 6.1, types of the form $\forall(X<:A)X\rightarrow B\{X\}$ (with $B\{X\}$ covariant in X) arise naturally for methods used in updating. In our type system, these types contain useful elements because of our structural assumptions. In contrast, they have no interesting elements in standard models of subtyping; see [8].

Types of this form can be used for defining classes as collections of pre-methods, where a pre-method is a procedure that is later used to construct a method. Each pre-method must work for all possible subclasses, parametrically in self, so that it can be inherited and instantiated to any of these subclasses. This is precisely what a type of the form $\forall(X<:A)X\rightarrow B\{X\}$ expresses.

We associate a class type Class(A) to each object type A. (We make the components of Class(A) invariant, for simplicity.)

If $\quad A \equiv \mathrm{Obj}(X)[l_i\upsilon_i{:}B_i\{X\}^{\,i\in 1..n}]$

then $\quad \mathrm{Class}(A) \triangleq [\mathrm{new}{:}A,\ l_i{:}\forall(X<:A)X\rightarrow B_i\{X\}^{\,i\in 1..n}]$.

A class c of type Class(A) consists of a particular collection of pre-methods $l_i$ of the appropriate types, along with a method, called new, for constructing new objects. The implementation of new is uniform for all classes: it produces an object of type A by collecting all the pre-methods of the class and applying them to the self of the new object.

$$c : \mathrm{Class}(A) \triangleq [\mathrm{new} = ς(z)\ [l_i{=}ς(x)\ z.l_i()(x)^{\,i\in 1..n}],\ l_1 = ..., \ldots, l_n = ...\ ]$$

The methods $l_i$ do not normally use the self of the class, but new does. For example:

Class(Mem) $\equiv$
[new: Mem,
get: $\forall(X<:\mathrm{Mem})\ X\rightarrow\mathrm{Bool}$,
set: $\forall(X<:\mathrm{Mem})\ X\rightarrow\mathrm{Bool}\rightarrow X$]

memClass: Class(Mem) $\triangleq$
  [new = ς(z) [get = ς(x) z.get()(x), set = ς(x) z.set()(x)],
   get = λ() λ(x) false,
   set = λ() λ(x) λ(b) x.get:=b]

m : Mem $\triangleq$ memClass.new

We can now consider the inheritance relation between classes. Suppose that we have another type $A' \equiv Obj(X)[l_i\upsilon'_i{:}B'_i\{X\}^{\;i\in 1..n+m}] <: A$, and a corresponding class type $Class(A') \equiv [new{:}A', l_i{:}\forall(X<:A')X\to B'_i\{X\}^{\;i\in 1..n+m}]$. For all $i\in 1..n$, we say that:

$l_i$ is inheritable from Class(A) to Class(A') iff $X<:A'$ implies $B_i\{X\}<:B'_i\{X\}$

When $l_i$ is inheritable, we have $\forall(X<:A)X\to B_i\{X\} <: \forall(X<:A')X\to B'_i\{X\}$. So, if c:Class(A) and $l_i$ is inheritable, we obtain $c.l_i : \forall(X<:A')X\to B'_i\{X\}$ by subsumption. Then, $c.l_i$ has the correct type to be reused when building a class c':Class(A'). That is, it can be inherited. For example, get and set are inheritable from Class(Mem) to Class(MemDup):

Class(MemDup) $\equiv$
  [new: MemDup,
   get: $\forall$(X<:MemDup) X→Bool,
   set: $\forall$(X<:MemDup) X→Bool→X,
   dup: $\forall$(X<:MemDup) X→X]

memDupClass: Class(MemDup) $\triangleq$
  [new = ς(z) [get = ς(x) z.get()(x), set = ς(x) z.set()(x), dup = ς(x) z.dup()(x)],
   get = memClass.get,
   set = memClass.set,
   dup = λ() λ(x) clone(x)]

The inheritability condition holds for any invariant component $l_i$, since in this case $B_i\{X\}\equiv B'_i\{X\}$. For contravariance, if $A' \equiv Obj(X)[l_i^-{:}B_i'\{X\}, ...]$ and $A \equiv Obj(X)[l_i^-{:} B_i\{X\}, ...]$ with $A'<:A$, then $X<:A'$ always implies $B_i\{X\}<:B_i'\{X\}$. Thus, inheritability of contravariant components is also guaranteed.

Covariant components do not necessarily correspond to inheritable pre-methods. For example, if $A' \equiv [l^+{:}Nat]$ and $A \equiv [l^+{:}Int]$ with Nat<:Int, and $c : [new{:}A, l{:}\forall(X<:A)X\to Int]$, then c.l cannot be inherited into $Class(A') \equiv [new{:}A', l{:}\forall(X<:A')X\to Nat]$, because it would produce a bad result. However, a class c' : Class(A') can include a different method for l with result type Nat: this corresponds to method specialization on overriding.

In conclusion, covariant components induce mild restrictions in subclassing. Invariant and contravariant components induce no restrictions. In practice, inheritability is expected between a class type C and another class type C' obtained as an extension of C (so that C' and C have identical common components). In this case, inheritability trivially holds for components of any variance.

## 7. Conclusions

We have described a basic calculus for objects and their types. It includes a sound type system with Self types within an imperative framework. Classes and inheritance arise from object types and polymorphic types. Because of its compactness and expressiveness, this calculus is appealing as a kernel for object-oriented languages that include subsumption and Self types.

## References

[1] Abadi, M., **Baby Modula-3 and a theory of objects**. *Journal of Functional Programming* **4**(2), 249-283. 1994.

[2] Abadi, M. and L. Cardelli, **A semantics of object types**. *Proc. IEEE Symposium on Logic in Computer Science*, 332-341. 1994.

[3] Abadi, M. and L. Cardelli, **A theory of primitive objects: second-order systems**. *Proc. ESOP'94 - European Symposium on Programming*. Springer-Verlag. 1994.

[4] Abadi, M. and L. Cardelli, **A theory of primitive objects: untyped and first-order systems**. *Proc. Theoretical Aspects of Computer Software*. Springer-Verlag. 1994.

[5] Abadi, M. and L. Cardelli, **An imperative object calculus: basic typing and soundness**. *Proc. Second ACM SIGPLAN Workshop on State in Programming Languages*, 19-32. Technical Report UIUCDCS-R-95-1900, University of Illinois at Urbana Champaign. 1995.

[6] Amadio, R.M. and L. Cardelli, **Subtyping recursive types**. *ACM Transactions on Programming Languages and Systems* **15**(4), 575-631. 1993.

[7] Bruce, K.B., **A paradigmatic object-oriented programming language: design, static typing and semantics**. *Journal of Functional Programming* **4**(2), 127-206. 1994.

[8] Bruce, K.B. and G. Longo, **A modest model of records, inheritance and bounded quantification**. *Information and Computation* **87**(1/2), 196-240. 1990.

[9] Bruce, K.B., A. Schuett, and R. van Gent, **PolyTOIL: a type-safe polymorphic object-oriented language.** Williams College. 1994.

[10] Cardelli, L., **Extensible records in a pure calculus of subtyping**. In *Theoretical Aspects of Object-Oriented Programming,* C.A. Gunter and J.C. Mitchell, ed. MIT Press. 373-425. 1994.

[11] Cardelli, L. and J.C. Mitchell, **Operations on records**. *Mathematical Structures in Computer Science* **1**(1), 3-48. 1991.

[12] Cardelli, L., J.C. Mitchell, S. Martini, and A. Scedrov, **An extension of system F with subtyping**. *Information and Computation* **109**(1-2), 4-56. 1994.

[13] Cook, W.R., **A proposal for making Eiffel type-safe**. *Proc. European Conference of Object-Oriented Programming*, 57-72. 1989.

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

[15] Harper, R., **A simplified account of polymorphic references**. *Information Processing Letters* **51**(4). 1994.

[16] Harper, R. and B. Pierce, **A record calculus based on symmetric concatenation**. *Proc. 18th Annual ACM Symposium on Principles of Programming Languages*. 1991.

[17] Leroy, X., **Polymorphic typing of an algorithmic language**. Rapport de Recherche no.1778 (Ph.D Thesis). INRIA. 1992.

[18] Meyer, B., **Object-oriented software construction**. Prentice Hall. 1988.

[19] Mitchell, J.C., F. Honsell, and K. Fisher, **A lambda calculus of objects and method specialization**. *Proc. 8th Annual IEEE Symposium on Logic in Computer Science*. 1993.

[20] Pierce, B.C. and D.N. Turner, **Simple type-theoretic foundations for object-oriented programming**. *Journal of Functional Programming* **4**(2), 207-247. 1994.

[21] Rémy, D., **Typechecking records and variants in a natural extension of ML**. *Proc. 16th Annual ACM Symposium on Principles of Programming Languages*, 77-88. 1989.

[22] Szypersky, C., S. Omohundro, and S. Murer, **Engineering a programming language: the type and class system of Sather**. TR-93-064. ICSI, Berkeley. 1993.

[23] Tofte, M., **Type inference for polymorphic references**. *Information and Computation* **89**, 1-34. 1990.

[24] Wand, M., **Type inference for record concatenation and multiple inheritance**. *Proc. 4th Annual IEEE Symposium on Logic in Computer Science*, 92-97. 1989.

[25] Wright, A.K. and M. Felleisen, **A syntactic approach to type soundness**. *Information and Computation* **115**(1), 38-94. 1994.

[26] Yonezawa, A. and M. Tokoro, ed. **Object-oriented concurrent programming**. MIT Press. 1987.