An Imperative Object Calculus

Martín Abadi and Luca Cardelli

Digital Equipment Corporation
Systems Research Center

FASE '95

Outline

Object calculi are formalisms at the same level of abstraction as λ-calculi, but based exclusively on objects rather than functions.

- An untyped object calculus.
- An imperative operational semantics.
- A type system.
  - Self types.
  - Variance annotations.
  - Structural subtyping assumptions.
  - Polymorphism.
- Classes and inheritance.
- Typing soundness, based on store typings.

New from last year:

- Imperative/operational semantics (instead of functional/denotational).
- Primitive Self type (instead of encoded).
- Primitive variance annotations (instead of encoded).
- Structural subtyping assumptions (which are not denotationally sound).
- Soundness based on subject reduction (rather than models).
- Class encodings, requiring polymorphism and structural subtyping assumptions.

Syntax and Informal Semantics

The evaluation of terms is based on an imperative operational semantics with a global store; it proceeds deterministically from left to 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
$a.l \Leftarrow \varsigma(x)b$	method update (imperative)
$let\ x = a\ in\ b$	let (sequential evaluation)
$clone(a)$	cloning (shallow copy)

- An object is a collection of components $l_i = \varsigma(x_i)b_i$, for distinct labels (method names) l_i and associated methods $\varsigma(x_i)b_i$. The methods are parameterless: x_i is a name for *self* within b_i.
- The letter ς (sigma) is a binder; it delays evaluation of the term to its right.

The let and method update constructs may be combined into a single construct, for more expressive typing (see FASE proceedings).

A Small Example

We define a memory cell with *get*, *set*, and *dup* (duplicate) components:

$[get = false,$	field
$set = \varsigma(self)\ \lambda(b)$	method with parameter
$\quad self.get := b,$	field update
$dup = \varsigma(self)$	
$\quad clone(self)]$	self-cloning

Some new constructions are used here:

- Procedures (λ), which can be encoded.
- Booleans, which can be encoded much as in the λ-calculus.
- Fields and field update, which can be desugared as follows:

$$\begin{array}{l} let\ y_1 = false \\ in\quad [get = \varsigma(self)\ y_1, \\ \qquad set = \varsigma(self)\ \lambda(b) \\ \qquad\qquad let\ y_2 = b\ self.get \Leftarrow \varsigma(self)\ y_2, \\ \qquad \ldots\ ] \end{array}$$

Procedures

Consider an imperative call-by-value λ-calculus that includes abstraction, application, and assignment to λ-bound variables. E.g.: $(\lambda(x)\ x{:=}x{+}1)(3)$ is a term yielding 4.

Translation of procedures

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

Low-level interpretation

- The translation of a procedure $\lambda(x)b$ is a stack frame with an uninitialized (divergent) argument slot (*arg*), and a initial program counter (*val*) that points to code accessing the argument slot through a frame pointer (x).
- The translation of a procedure call allocates a fresh stack frame (by *clone*), fills the argument slot (by :=), and jumps to the code (by *.val*).

Operational Semantics

The semantics relates terms to results in a global store.

Notation

ι			store location (e.g., an integer)
v	::=	$[l_i{=}\iota_i\ ^{i\in 1..n}]$	object result(l_i distinct)
σ	::=	$\iota_i \mapsto \langle\varsigma(x_i)b_i, S_i\rangle\ ^{i\in 1..n}$	store for closures(ι_i distinct)
S	::=	$x_i \mapsto v_i\ ^{i\in 1..n}$	stack for results (x_i distinct)

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

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

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

Sample rules

(Red Object) (l_i, ι_i distinct)

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

(Red Select)

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

(Red Simple Update)

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

N.B. The term:

$$[l = \varsigma(x)\ x.l := x].l$$

creates a loop in the store. An attempt to read out the result by "inlining" the store and stack mappings would produce the infinite term:

$$[l{=}\varsigma(x)[l{=}\varsigma(x)[l{=}\varsigma(x)\ldots]]]$$

A Type System

We develop a type system for the imperative calculus. We treat Self types, variance annotations, and structural subtyping assumptions. Simpler (and less expressive) type systems could also be defined.

Syntax of types

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

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

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

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

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

Self Types

Intent: memory cells can be typed as:

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

In general, let:

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

- A is the type of those objects with methods named l_i and result types $B_i\{\!\{A\}\!\}$.
- The binder Obj binds a Self type named X (which is known to be a subtype of A).

Moreover:

- The υ_i are variance annotations.
- The variable X may occur only covariantly in the types B_i.

Notation

- $B\{X\}$ means that X may occur free in B.
- $B\{X^+\}$ means that X occurs covariantly in B.
- $B\{\!\{A\}\!\}$ is the result of substituting A for X in $B\{X\}$, where X is clear from context.

The subtyping rule for object types with Self asserts, as usual, that a "longer" object type is a subtype of a "shorter" one.

A simplified rule for object types without variance annotations reads:

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

For example:

$$Mem \triangleq Obj(X)[get{:}\ Bool,\ set{:}\ Bool \to X]$$
$$MemDup \triangleq Obj(X)[get{:}\ Bool,\ set{:}\ Bool \to X,\ dup{:}\ X]$$
$$MemDup <: Mem$$

The type $Obj(X)[...]$ can be viewed as a recursive type $\mu(X)[...]$, but with differences in subtyping that are crucial for object-oriented applications. The subtyping rule above is unsound with recursive types instead of Self types (i.e. with μ instead of Obj), in presence of subsumption and update.

Note: Counterexample

$memDup : MemDup \triangleq$
$\quad [get = \varsigma(self)\ let\ x = self.set(false).dup\ in\ false,$
$\quad set = \varsigma(self)\ \lambda(b)\ self,$
$\quad dup = \varsigma(self)\ self\ ]$

$mem : Mem \triangleq memDup$ $\qquad$ since $MemDup <: Mem$

$mem.set := \lambda(b)\ [get = false,\ set = \varsigma(self)\ \lambda(b)\ self]$

$mem \equiv$
$\quad [get = \varsigma(self)\ let\ x = self.set(false).dup\ in\ false,$
$\quad set = \varsigma(self)\ \lambda(b)\ [get = false,\ set = \varsigma(self)\ \lambda(b)\ self],$
$\quad dup = \varsigma(self)\ self\ ]$

$mem.get$ $\qquad$ FAILS!

Variance Annotations

Again, let:

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

Each υ_i is a variance annotation; it is one of the symbols $^-$, $^\circ$, and $^+$, for contravariance, invariance, and covariance, respectively.

Intuitively, $^+$ means read-only, $^-$ means write-only, and $^\circ$ means read-write.

- $^+$ prevents update, but allows covariant component subtyping.
- $^-$ prevents invocation, but allows contravariant component subtyping.
- $^\circ$ allows both invocation and update, but requires exact matching in subtyping.

By convention, any omitted υ's are taken to be equal to $^\circ$.

A simple object type:

$$[l_i{:}B_i\;^{i\in 1..n}]$$

is an abbreviation for $Obj(X)[l_i^\circ{:}B_i\;^{i\in 1..n}]$, where X does not appear in any B_i.

Variance Rules

Because of variance annotations, we use an auxiliary subtyping judgment:

(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)

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

(Sub Covariant)

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

(Sub Contravariant)

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

- (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.

Example: Procedure Types

A procedure with argument of type A and result of type B, encoded as shown earlier, can be given type:

$$[arg^\circ{:}\, A,\; val^\circ{:}\, B]$$

By the subtyping rules for variances we obtain:

$$[arg^\circ{:}\, A,\; val^\circ{:}\, B] \;\;<:\;\; [arg^-{:}\, A,\; val^+{:}\, B]$$

By subsumption, any procedure has the type on the right. Therefore, we can take:

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

Which yields a defined notion of procedure type that is contravariant in the argument and covariant in the result type.

Example: State Encapsulation

One can hide certain object components from view simply by subsumption; this technique can be used to encapsulating state.

Variance annotations enable more sophisticated forms of encapsulation.

$$Mem \triangleq Obj(X)[get^\circ{:}Bool,\; set^\circ{:}Bool\to X]$$

$$mem : Mem \triangleq [get = false,\; set = \varsigma(self)\,\lambda(b)\, self.get := b]$$

N.B. *get* is both read and written

When considering a memory cell as an object encapsulating state, it is natural to expect both components of *Mem* to be protected against external update. Take:

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

Since $Mem <: ProtectedMem$, any memory cell can be subsumed into *ProtectedMem* and thus protected against updating from the outside.

Note that the *set* method can still update the *get* field "from the inside".

Polymorphism

Additional syntax of terms

a,b	::=	term
	. . .	(as before)
	$\lambda()b$	type abstraction
	$a()$	type application

N.B. $\lambda()b$ is the type-erasure of $\lambda(X<:A)b$; $a()$ is the type-erasure of $a(A)$.

Additional results

v	::=	result
	. . .	(as before)
	$\langle\lambda()b,S\rangle$	type abstraction result

Additional term reductions (...)

Additional syntax of types

A,B	::=	type
	...	(as before)
	$\forall(X<:A)B$	bounded universal quantifier

Additional typing rules (...)

Structural Subtyping Assumptions

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

$$\frac{E \vdash a : A \qquad E, Y<:A \vdash b : B_j\{\!|Y|\!\} \qquad \upsilon_j\in\{^\circ,^-\} \qquad j\in 1..n}{E \vdash a.l_j{:=}b : A}$$

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

$$\frac{E \vdash a : A \qquad E \vdash A<:A' \qquad E, Y<:A \vdash b : B_j\{\!|Y|\!\} \qquad \upsilon_j\in\{^\circ,^-\} \qquad j\in 1..n}{E \vdash a.l_j{:=}b : A}$$

$Mem \triangleq Obj(X)[get^\circ{:}Bool,\ set^\circ{:}Bool\rightarrow X]$

E, X<:Mem, x:X, b:Bool $\vdash$ x : X
E, X<:Mem, x:X, b:Bool $\vdash$ X <: Mem
E, X<:Mem, x:X, b:Bool $\vdash$ b : Bool
E, X<:Mem, x:X, b:Bool $\vdash$ x.get:=b : X

E, X<:Mem $\vdash$ λ(x) λ(b) x.get:=b : X→Bool→X
$E \vdash \lambda()\ \lambda(x)\ \lambda(b)\ x.get{:=}b\ :\ \forall(X<:Mem)\ X\rightarrow Bool\rightarrow X$

N.B. We have obtained a non-trivial term of type $\forall(X<:Mem)\ X\rightarrow B\{\!|X|\!\}$. The non-structural rule would only yield $\forall(X<:Mem)\ X\rightarrow B\{\!|Mem|\!\}$.

Classes as Collections of Pre-Methods

We define classes as collections of reusable pre-methods.

- A pre-method is a procedure that is later used to construct a method.
- Each pre-method must work for all possible subclasses of a given class, so that it can be inherited and instantiated to any of these subclasses.
- To this end, pre-methods have types of the form $\forall(X<:A)X\rightarrow B_i\{X\}$.

We associate a class type $Class(A)$ to each object type A:

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

The implementation of *new* is uniform for all classes: it produces an object of type A by collecting all the pre-methods and applying them to the self of the new object.

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

Classes

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

$memClass{:}\ Class(Mem) \triangleq$
 $[new = \varsigma(z)\ [get = \varsigma(x)\ z.get()(x),\ set = \varsigma(x)\ z.set()(x)],$
 $get = \lambda()\ \lambda(x)\ false,$
 $set = \lambda()\ \lambda(x)\ \lambda(b)\ x.get{:=}b]$

$m : Mem \triangleq memClass.new$

Note that the *set* pre-method receives the desired type (as shown earlier) thanks to the structural subtyping assumptions.

Subclasses and Inheritance

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

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

Note that:

- $memClass.set : \forall(X<:Mem)X \to Bool \to X$
- $\forall(X<:Mem)X \to Bool \to X \ <:\ \forall(X<:MemDup)X \to Bool \to X$
- by subsumption, $memClass.set : \forall(X<:MemDup)X \to Bool \to X$
- therefore, *memClass.set* can be reused as a pre-method of *Class*(*MemDup*).

Soundness

Store types

$M ::= Obj(X)[l_i \upsilon_i : B_i\{X\}^{\ i \in 1..n}] \Rightarrow j$	method type ($j \in 1..n$)
$\Sigma ::= \iota_i \mapsto M_i^{\ i \in 1..n}$	store type (ι_i distinct)

Type stacks

$T \equiv X_i \mapsto A_i^{\ i \in 1..n}$	type stack (A_i closed types)

Result typing judgment: $\Sigma \vDash v : A$ (A closed)

Stack typing judgment: $\Sigma \vDash S{\bullet}T : E$

Store typing judgment: $\Sigma \vDash \sigma$

N.B. The fact that values are typed with respect to store types (and not stores) allows us to deal with cycles in the store.

Theorem (Subject Reduction)

If $\varnothing \vdash a : A$ and $\varnothing{\bullet}\varnothing \vdash a \leadsto v{\bullet}\sigma$
then there exist a type $A^\dagger$ and a store type $\Sigma^\dagger$ such that
$\Sigma^\dagger \vDash \sigma$ and $\Sigma^\dagger \vDash v : A^\dagger$, with $\varnothing \vdash A^\dagger <: A$.

□

Conclusions

- We have described a basic calculus for imperative objects and their types.
- Because of its compactness and expressiveness, this calculus is appealing as a kernel for object-oriented languages that include subsumption and Self types.
- The calculus is not class-based, since classes are not built-in, nor delegation-based, since the method-lookup mechanism does not delegate invocations. However, the calculus models class-based languages well: classes and inheritance arise from object types and polymorphic types. In delegation-based languages, traits play the role of classes; our calculus can model traits just as easily as classes, along with dynamic delegation based on traits.