A Theory of Primitive Objects

Martín Abadi and Luca Cardelli

Digital Equipment Corporation, Systems Research Center

CONTENTS

INTRODUCTION

- Various λ-calculi have been used successfully as foundations for procedural programming.
- Object-oriented programming is advertised as a new computing paradigm, conceptually and pragmatically separate form ordinary procedural programming.
- Is that true? That is, are there *object calculi*, that can play the role of λ-calculi for object-oriented programming?

- The immediate answer seems *no*, or *why bother*. After all, methods have parameters, and parameters need to be modeled by λ-abstraction. Hence object calculi, whatever they are, must be extensions of λ-calculi. Functions seem more fundamental than objects.
- This kind of answer has stimulated work aimed to encode object calculi within λ-calculi. This approach has been successful for untyped calculi, but has proven unexpectedly difficult for typed calculi.

Approach

- We aim to investigate the type theory of object calculi. That is, we aim to investigate the properties of object types separately from the properties of function types. We are pushed in this direction because natural notions of object types are not easily, or at all, definable in most standard formalism.
- In the process, we discover that object calculi can be seen even as *more fundamental* than λ-calculi: functions can be defined from objects, and function types can be defined from object types, in a uniform way and in a variety of formalisms. (The opposite is not nearly as easy.)
- For our study of object typing, it does not much matter whether the underlying computation paradigm is procedural, imperative, or concurrent. For easier comparison with λ-calculi we adopt a procedural (non-imperative, non-concurrent) semantics.

A <u>Very Informal</u> Tutorial

- **Object** ≈ data + operations ≈ fields + methods (Jargon)
  "A vehicle with seats and a method of locomotion." (Concept)
  A VW with a VW engine. (Instance)
- **Object Subsumption**
  "This object is as good as that object, use this one instead."
  Drive a Porsche, not a VW.
- **Method Override** (i.e. replacement)
  "This method is as good as that method; use this on instead."
  Replace this car's engine with an old VW engine.
- **Subsumption + Override = Problems**
  "Replace this object with a better one, then replace its methods."
  Replace this Porche's engine with an old VW engine. (!!)

Solutions:

- **Subsumption without Override**
  Drive a Porche; <u>never change its engine</u>.
- **Override without Subsumption**
  Install an old VW engine, <u>but only in a VW</u>.
- **Parametric Override**
  "This method is as good as that method, <u>and it is fully general</u>; use this one instead."
  Replace <u>this car's</u> engine with an old engine <u>of the same kind</u>.

UNTYPED OBJECT CALCULI

An Untyped Object Calculus

Syntax of the ς-calculus

a,b ::=	terms
x	variable
$[l_i=\varsigma(x_i)b_i \; ^{i\in 1..n}]$ (l_i distinct)	object
a.l	field selection / method invocation
$a.l \Leftarrow \varsigma(x)b$	field update / method override

λ(x)b	function
b(a)	application

An **object** $[l_i=\varsigma(x_i)b_i \; ^{i\in 1..n}]$ is a collection of **methods** $\varsigma(x_i)b_i$, named by l_i.
A method ς(x)b has a **self** parameter x, and a **body** b.
A method's self denotes its **host object**.
Through self, a method may refer to its **sibling methods**.

Primitive Semantics of the ς-calculus

Let $o \equiv [l_i=\varsigma(x_i)b_i\ ^{i\in 1..n}]$			(l_i distinct)
$o.l_j$	$\rightsquigarrow$	$b_j\{x_j \leftarrow o\}$	($j\in 1..n$)
$o.l_j \Leftarrow \varsigma(y)b$	$\rightsquigarrow$	$[l_j=\varsigma(y)b,\ l_i=\varsigma(x_i)b_i\ ^{i\in(1..n)-\{j\}}]$	($j\in 1..n$)

Theorem: Church-Rosser

We are dealing with a calculus of objects (not of functions), with a deterministic semantics that is non-imperative and non-concurrent.

We have investigated imperative versions of the calculus.

We have not investigated a concurrent version.

(Note)

Equational Theory of the ς-calculus

(Eq Symm)
$$\frac{b \leftrightarrow a}{a \leftrightarrow b}$$

(Eq Trans)
$$\frac{a \leftrightarrow b \qquad b \leftrightarrow c}{a \leftrightarrow c}$$

(Eq x)
$$\frac{}{x \leftrightarrow x}$$

(Eq Object) (l_i distinct)
$$\frac{b_i \leftrightarrow b_i' \quad \forall i\in 1..n}{[l_i=\varsigma(x_i)b_i\ ^{i\in 1..n}] \leftrightarrow [l_i=\varsigma(x_i)b_i'\ ^{i\in 1..n}]}$$

(Eq Select)
$$\frac{a \leftrightarrow a'}{a.l \leftrightarrow a'.l}$$

(Eq Override)
$$\frac{a \leftrightarrow a' \qquad b \leftrightarrow b'}{a.l\Leftarrow\varsigma(x)b \leftrightarrow a'.l\Leftarrow\varsigma(x)b'}$$

(Eval Select) (where $a\equiv[l_i=\varsigma(x_i)b_i\ ^{i\in 1..n}]$)
$$\frac{j\in 1..n}{a.l_j \leftrightarrow b_j\{x_j\leftarrow a\}}$$

(Eval Override) (where $a\equiv[l_i=\varsigma(x_i)b_i\ ^{i\in 1..n}]$)
$$\frac{j\in 1..n}{a.l_j\Leftarrow\varsigma(x)b \leftrightarrow [l_j=\varsigma(x)b,\ l_i=\varsigma(x_i)b_i\ ^{i\in(1..n)-\{j\}}]}$$

Basic Examples

Let $o \triangleq [l=\varsigma(x)x.l]$ A divergent method.
then $o.l \rightsquigarrow x.l\{x\leftarrow o\} \equiv o.l \rightsquigarrow \ ...$

Let $o' \triangleq [l = \varsigma(x)x]$ A self-returning method.
then $o'.l \rightsquigarrow x\{x\leftarrow o'\} \equiv o'$

Let $o'' \triangleq [l = \varsigma(y)\,(y.l\Leftarrow\varsigma(x)x)]$ A self-modifying method.
then $o''.l \rightsquigarrow (o''.l\Leftarrow\varsigma(x)x) \rightsquigarrow o'$

Notation

Our calculus is based entirely on methods; fields (instance variables) can be seen as methods that do not user their self parameter:

Fields:

$[..., l=b, ...] \triangleq [..., l=\varsigma(y)b, ...]$, for an unused y

Field update:

$o.l_j:=b \triangleq o.l_j\Leftarrow\varsigma(y)b$, for an unused y

Moreover, it is convenient to have a notation for “uninitialized” components, which are set to diverge on invocation:

$[..., l\!\uparrow, ...] \triangleq [..., l=\varsigma(x)x.l, ...]$

Example: Object-Oriented Booleans

$$\begin{array}{lll}
\text{true} & \triangleq & [\text{if} = \varsigma(x)\, x.\text{then},\ \ \text{then}\uparrow,\ \ \text{else}\uparrow] \\
\text{false} & \triangleq & [\text{if} = \varsigma(x)\, x.\text{else},\ \ \text{then}\uparrow,\ \ \text{else}\uparrow] \\
\text{cond}(b,a',a'') & \triangleq & ((b.\text{then}{:=}a').\text{else}{:=}a'').\text{if}
\end{array}$$

$$\begin{array}{ll}
\text{cond}(\text{true, false, true}) \ \equiv & ((\text{true.then}{:=}\text{false}).\text{else}{:=}\text{true}).\text{if} \\
\leadsto & ([\text{if} = \varsigma(x)\, x.\text{then},\ \ \text{then} = \text{false},\ \ \text{else}\uparrow].\text{else}{:=}\text{true}).\text{if} \\
\leadsto & [\text{if} = \varsigma(x)\, x.\text{then},\ \ \text{then} = \text{false},\ \ \text{else} = \text{true}].\text{if} \\
\leadsto & [\text{if} = \varsigma(x)\, x.\text{then},\ \ \text{then} = \text{false},\ \ \text{else} = \text{true}].\text{then} \\
\leadsto & \text{false}
\end{array}$$

Example: Object-Oriented Naturals

$$\begin{array}{l}
\text{zero} \triangleq \\
\quad [\text{case} = \lambda(z)\, \lambda(s)\, z, \\
\quad\ \ \text{succ} = \varsigma(x)\, x.\text{case} := \lambda(z)\, \lambda(s)\, s(x)\ ]
\end{array}$$

Each numeral has a case fields that contains either $\lambda(z)\lambda(s)z$ for zero, or $\lambda(z)\lambda(s)s(x)$ for non-zero, where x is the predecessor (self). Each numeral has a succ fields that modifies the case field to the non-zero version.

Informally: $n.\text{case}(z)(s) \ = \ z$ if n is zero, $s(n\text{-}1)$ otherwise

$$\begin{array}{lllll}
\text{zero} & & \equiv & [\text{case} = \lambda(z)\, \lambda(s)\, z, & \text{succ} = \ldots\ ] \\
\text{one} & \triangleq\ \text{zero.succ} & \equiv & [\text{case} = \lambda(z)\, \lambda(s)\, s(\text{zero}), & \text{succ} = \ldots\ ] \\
\text{two} & \triangleq\ \text{one.succ} & \equiv & [\text{case} = \lambda(z)\, \lambda(s)\, s(\text{one}), & \text{succ} = \ldots\ ]
\end{array}$$

$$\begin{array}{lll}
\text{iszero} & \triangleq & \lambda(n)\, n.\text{case}(\text{true})(\lambda(p)\text{false}) \\
\text{pred} & \triangleq & \lambda(n)\, n.\text{case}(\text{zero})(\lambda(p)p)
\end{array}$$

Example: Calculator

$$\begin{array}{l}
\text{calculator} \triangleq \\
\quad [\text{arg} = 0.0, \\
\quad\ \ \text{acc} = 0.0, \\
\quad\ \ \text{enter} = \varsigma(s)\, \lambda(n)\, s.\text{arg} := n, \\
\quad\ \ \text{add} = \varsigma(s)\, (s.\text{acc} := s.\text{equals}).\text{equals} \Leftarrow \varsigma(s')\, s'.\text{acc}{+}s'.\text{arg}, \\
\quad\ \ \text{sub} = \varsigma(s)\, (s.\text{acc} := s.\text{equals}).\text{equals} \Leftarrow \varsigma(s')\, s'.\text{acc}{-}s'.\text{arg}, \\
\quad\ \ \text{equals} = \varsigma(s)\, s.\text{arg}\ ]
\end{array}$$

The equals methods works as the result button and as the operator stack.

calculator .enter(5.0) .equals	=	5.0
calculator .enter(5.0) .sub .enter(3.5) .equals	=	1.5
calculator .enter(5.0) .add .add .equals	=	15.0

Functions as Objects

Translation of the untyped λ-calculus

$$\begin{array}{l}
\langle\!\langle x \rangle\!\rangle \triangleq x \\
\langle\!\langle b(a) \rangle\!\rangle \triangleq \langle\!\langle b \rangle\!\rangle \bullet \langle\!\langle a \rangle\!\rangle \qquad\qquad \text{where } \ p \bullet q \triangleq (p.\text{arg} := q).\text{val} \\
\langle\!\langle \lambda(x)b \rangle\!\rangle \triangleq \\
\quad [\text{arg}\uparrow, \\
\quad\ \ \text{val} = \varsigma(x)\langle\!\langle b \rangle\!\rangle\{x \leftarrow x.\text{arg}\}]
\end{array}$$

$$\langle\!\langle (\lambda(x)b)(a) \rangle\!\rangle \ \equiv\ ([\text{arg}\uparrow,\ \ \text{val} = \varsigma(x)\langle\!\langle b \rangle\!\rangle\{x \leftarrow x.\text{arg}\}].\text{arg}{:=}\langle\!\langle a \rangle\!\rangle).\text{val} \ \leadsto\ \langle\!\langle b\{x \leftarrow a\} \rangle\!\rangle$$

Preview: this encoding extends to typed calculi:

$\langle\!\langle A \to B \rangle\!\rangle \triangleq [\text{arg}: \langle\!\langle A \rangle\!\rangle, \text{val}: \langle\!\langle B \rangle\!\rangle]$		1st-order λ into 1st-order ς
$V \triangleq \mu(X)[\text{arg}: X, \text{val}: X]$		untyped λ into 1st-order ς with μ

Recursion

$$\langle\!\langle \mu(x)b \rangle\!\rangle \triangleq [rec = \varsigma(x)\langle\!\langle b \rangle\!\rangle\{x \leftarrow x.rec\}].rec$$

$$\langle\!\langle \mu(x)b \rangle\!\rangle \equiv [rec = \varsigma(x)\langle\!\langle b \rangle\!\rangle\{x \leftarrow x.rec\}].rec \rightsquigarrow \langle\!\langle b\{x \leftarrow \mu(x)b\} \rangle\!\rangle$$

$$fix \triangleq [arg\uparrow,\ val = \varsigma(x)((x.arg).arg := x.val).val]$$

$$fix_f \triangleq fix.arg := f \rightsquigarrow [arg = f,\ val = \varsigma(x)((x.arg).arg := x.val).val]$$

$$fix \bullet f \rightsquigarrow fix_f.val \rightsquigarrow ((fix_f.arg).arg := fix_f.val).val \rightsquigarrow (f.arg := fix \bullet f).val$$
$$\equiv f \bullet (fix \bullet f)$$

Towards Types for Untyped Calculi

A **record type**

$$\langle l_i{:}B_i \ ^{i \in 1..n} \rangle$$

is the type of records with components named l_i of type B_i.

A record with more components is a **subtype** of a record with fewer component, provided the corresponding components are in the subtype relation:

$$\langle l_i{:}B_i \ ^{i \in 1..n+m} \rangle <: \langle l_i{:}B'_i \ ^{i \in 1..n} \rangle \quad \text{if} \quad B_i <: B'_i \quad \text{for } i \in 1..n$$

Record types are **covariant** in their components.

An record can be used in place of another record with fewer methods, by **subsumption**:

$$a{:}A \quad \wedge \quad A{<:}B \quad \Rightarrow \quad a{:}B$$

An **object type**

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

is the type of those objects with methods l_i: $\varsigma(x{:}A)b_i$, each having self type $A <: [l_i{:}B_i \ ^{i \in 1..n}]$ with result type B_i.

An object type with more methods is a **subtype** of an object type with fewer methods:

$$[l_i{:}B_i \ ^{i \in 1..n+m}] <: [l_i{:}B_i \ ^{i \in 1..n}]$$

Object types are **invariant** (neither covariant nor contravariant) in their components. This is necessary for soundness.

An object can be used in place of another object with fewer methods, by **subsumption**:

$$a{:}A \quad \wedge \quad A{<:}B \quad \Rightarrow \quad a{:}B$$

Interpretations

Self-Application Semantics [Kamin 1988]

$$\begin{array}{llll} [l_i{=}\varsigma(x_i)b_i \ ^{i \in 1..n}] & \triangleq & \langle l_i{=}\lambda(x_i)b_i \ ^{i \in 1..n} \rangle & \text{(a record of functions)} \\ o.l_j & \triangleq & o \cdot l_j(o) \quad \rightsquigarrow \quad b_j\{x_j \leftarrow o\} & (j \in 1..n) \\ o.l_j \Leftarrow \varsigma(y)b & \triangleq & o \cdot l_j{:=}\lambda(y)b \quad \rightsquigarrow \quad [l_j{=}\varsigma(y)b,\ l_i{=}\varsigma(x_i)b_i \ ^{i \in (1..n)-\{j\}}] & (j \in 1..n) \end{array}$$

This is a valid interpretation (it satisfies the primitive semantics). Moreover, it matches the most common implementation technique for objects, so it is the *intended* interpretation.

But it does not extend to typed calculi:

$$[l_i{:}B_i \ ^{i \in 1..n}] \triangleq \mu(X)\langle l_i{:}X \rightarrow B_i \ ^{i \in 1..n} \rangle$$

But NOT, e.g.: $\mu(X)\langle l{:}X \rightarrow A,\ l'{:}X \rightarrow B \rangle <: \mu(Y)\langle l{:}Y \rightarrow A \rangle$

So, regrettably, this interpretation cannot be adopted.

N.B. we provide a denotational semantics in CUPER, which is based on the self-application interpretation.

Recursive-Records Semantics [Cardelli 1988]

$$[l_i{:}B_i\ ^{i\in 1..n}] \triangleq \langle l_i{:}B_i\ ^{i\in 1..n}\rangle$$
$$[l_i{=}\varsigma(x_i)b_i\ ^{i\in 1..n}] \triangleq \mu(x)\langle l_i{=}b_i\{x_i{\leftarrow}x\}\ ^{i\in 1..n}\rangle$$

Satisfies object subtyping and the primitive semantics of invocation, but not the primitive semantics of override.

Generator Semantics [Cook 1989]

$[[l_i{:}B_i\ ^{i\in 1..n}]]$	$\triangleq$	$[l_i{:}B_i\ ^{i\in 1..n}]\rightarrow[l_i{:}B_i\ ^{i\in 1..n}]$	a generator type
$[l_i{:}B_i\ ^{i\in 1..n}]$	$\triangleq$	$\langle l_i{:}B_i\ ^{i\in 1..n}\rangle$	an object type
$[l_i{=}\varsigma(x_i)b_i\ ^{i\in 1..n}]$	$\triangleq$	$\lambda(x)\langle l_i{=}b_i\{x_i{\leftarrow}x\}\ ^{i\in 1..n}\rangle$	an object generator
new(g)	$\triangleq$	$\mathbf{Y}(g)$	an object

Satisfies object subtyping (not generator subtyping). Does not satisfy the primitive semantics of method invocation on objects.

Class Semantics [Pierce, Turner 1994]

$$[l_i{:}B_i\ ^{i\in 1..n}] \triangleq \exists(X)\langle l_i{:}X\rightarrow B_i\ ^{i\in 1..n}\rangle$$

Satisfies object subtyping but not the primitive semantics of override, because it separates fields from methods, and methods cannot be overridden in objects. A translation of objects is possible but non-trivial (is type-driven), because of the splitting of self into fields and methods. Fits well with class-based object-oriented languages.

Sum-of-Extensions Semantics [Abadi, Cardelli 1994b]

$$[l_i{:}B_i\ ^{i\in 1..n}] \triangleq \exists(X{::}{\uparrow}(l_i\ ^{i\in 1..n}))\ \mu(Y)\ Y\rightarrow\langle l_i{:}B_i\ ^{i\in 1..n},X\rangle$$

Derived from a denotational semantics of object types. Satisfies the primitive semantics (is based on the self-application semantics at the term level). Satisfies object subtyping only in a specialized theory of $\exists$.

- The interpretations that work best, if at all, are the most complicated. This is a pity, because the intended type theory of objects, which we now study, is almost trivial.

FIRST-ORDER OBJECT CALCULI

A First-order Calculus

Judgments:

$E \vdash \diamond$	environment E is well-formed
$E \vdash A$	A is a type in E
$E \vdash A <: B$	A is a subtype of B in E
$E \vdash a : A$	a has type A in E

Environments:

$E \equiv x_i{:}A_i\ ^{i\in 1..n}$	environments, with x_i distinct

Types:

$A,B\ ::=\ [l_i{:}B_i\ ^{i\in 1..n}]$	object types, with l_i distinct

Terms (identified up to α-conversion):

$$\begin{array}{ll} a,b\ ::= & x \\ & [l_i{=}\varsigma(x_i{:}A_i)b_i\ ^{i\in 1..n}] \\ & a.l \\ & a.l \Leftarrow \varsigma(x{:}A)b \end{array}$$

The object fragment:

(Type Object) (l_i distinct)

$$\frac{E \vdash B_i \qquad \forall i \in 1..n}{E \vdash [l_i{:}B_i \;^{i\in 1..n}]}$$

(Sub Object) (l_i distinct)

$$\frac{E \vdash B_i \qquad \forall i \in 1..n{+}m}{E \vdash [l_i{:}B_i \;^{i\in 1..n+m}] <: [l_i{:}B_i \;^{i\in 1..n}]}$$

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

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

(Val Select)

$$\frac{E \vdash a : [l_i{:}B_i \;^{i\in 1..n}] \qquad j \in 1..n}{E \vdash a.l_j : B_j}$$

(Val Override) (where $A \equiv [l_i{:}B_i \;^{i\in 1..n}]$)

$$\frac{E \vdash a : A \qquad E, x{:}A \vdash b : B_j \qquad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x{:}A)b : A}$$

With some additional, standard rules we obtain a complete calculus:

(Env $\emptyset$)

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

(Env x)

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

(Sub Refl)

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

(Sub Trans)

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

(Val x)

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

(Val Subsumption)

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

Theorem (Ob$_{1<:}$ has minimum types)

In **Ob$_{1<:}$**, if $E \vdash a : A$ then there exists B such that $E \vdash a : B$ and, for any A', if $E \vdash a : A'$ then $E \vdash B <: A'$.

Unsoundness of Covariance

$U \triangleq []$	The unit object type.
$L \triangleq [l{:}U]$	An object type with just l.
$L <: U$	

$P \triangleq [x{:}U, f{:}U]$

$Q \triangleq [x{:}L, f{:}U]$

Assume $Q <: P$ by an (erroneous) covariant rule for object subtyping.

$q : Q \triangleq [x{=}[l{=}[]], f{=}\varsigma(s{:}Q)s.x.l]$

then $q : P$ by subsumption with $Q <: P$

hence $q.x{:=}[] : P$ that is $[x{=}[], f{=}\varsigma(s{:}Q)s.x.l] : P$

But $(q.x{:=}[]).f$ fails!

(The essence of this counterexample is used to show the unsoundness of record type covariance in presence of side-effecting field update. Interestingly, with methods, the counterexample can be adapted to our side-effects-free setting.)

A First-order Equational Theory

Consider:

A	$\triangleq$	$[x{:}Nat, f{:}Nat]$
$a{:}A$	$\triangleq$	$[x{=}1, f{=}\varsigma(s{:}A)1]$
$b{:}A$	$\triangleq$	$[x{=}1, f{=}\varsigma(s{:}A)s.x]$

We have, informally, $a.x = b.x : Nat$ and $a.f = b.f : Nat$.

So, is a = b? Consider the context:

$C\{o\} \triangleq (o.x{:=}2).f$

We have $C\{a\} = 1 \neq 2 = C\{b\}$. Hence:

$a \neq b : A$

Still, $a = [x{=}1] : [x{:}Nat]$ and $b = [x{=}1] : [x{:}Nat]$. Hence:

$a = b : [x{:}Nat]$

Finally:

$a \overset{?}{=} b : [f{:}Nat]$

This is sound but not provable in our theory. It would be unsound in an imperative or concurrent context.

(Eq Object) (where $A\equiv[l_i{:}B_i\ ^{i\in1..n}]$)

$$\frac{E, x_i{:}A \vdash b_i \leftrightarrow b_i' : B_i \quad \forall i\in 1..n}{E\vdash [l_i=\varsigma(x_i{:}A)b_i\ ^{i\in1..n}] \leftrightarrow [l_i=\varsigma(x_i{:}A)b_i'\ ^{i\in1..n}] : A}$$

(Eq Sub Object) (where $A\equiv[l_i{:}B_i\ ^{i\in1..n}]$, $A'\equiv[l_i{:}B_i\ ^{i\in1..n}, l_j{:}B_j\ ^{j\in n+1..m}]$)

$$\frac{E, x_i{:}A \vdash b_i : B_i \quad \forall i\in 1..n \qquad E, x_j{:}A' \vdash b_j : B_j \quad \forall j\in n+1..m}{E\vdash [l_i=\varsigma(x_i{:}A)b_i\ ^{i\in1..n}] \leftrightarrow [l_i=\varsigma(x_i{:}A')b_i\ ^{i\in1..n+m}] : A}$$

(Eval Select) (where $A\equiv[l_i{:}B_i\ ^{i\in1..n}]$, $a\equiv[l_i=\varsigma(x_i{:}A')b_i\ ^{i\in1..n+m}]$)

$$\frac{E\vdash a : A \quad j\in 1..n}{E\vdash a.l_j \leftrightarrow b_j\{x_j\leftarrow a\} : B_j}$$

(Eval Override) (where $A\equiv[l_i{:}B_i\ ^{i\in1..n}]$, $a\equiv[l_i=\varsigma(x_i{:}A')b_i\ ^{i\in1..n+m}]$)

$$\frac{E\vdash a : A \quad E, x{:}A \vdash b : B_j \quad j\in 1..n}{E\vdash a.l_j \Leftarrow \varsigma(x{:}A)b \leftrightarrow [l_j=\varsigma(x{:}A')b, l_i=\varsigma(x_i{:}A')b_i\ ^{i\in(1..n+m)-\{j\}}] : A}$$

A	$\triangleq$	$[x{:}Nat, f{:}Nat]$
$a{:}A$	$\triangleq$	$[x=1, f=\varsigma(s{:}A)1]$
$b{:}A$	$\triangleq$	$[x=1, f=\varsigma(s{:}A)s.x]$

The rule (Eq Object) is a congruence rule. (We omit the obvious congruence rules for selection and override.) It can only compare objects of equal length. Hence this is not sufficient to prove $a\leftrightarrow[x=1]{:}[x{:}Nat]$.

Objects of different lengths can be compared by (Eq Sub Object). This rule requires that, in the longer object, the common methods do not depend on the additional methods, so that the common methods can be typed with the shorter type as the type of self.

This allows us to conclude $a\leftrightarrow[x=1]{:}[x{:}Nat]$, but not $a\leftrightarrow[f=\varsigma(s{:}A)s.x]{:}[f{:}Nat]$, because f depends on a hidden method

Function Types

Translation of function types:

$\langle\!\langle A\to B\rangle\!\rangle \triangleq [arg{:}\langle\!\langle A\rangle\!\rangle, val{:}\langle\!\langle B\rangle\!\rangle]$

$\langle\!\langle x_A\rangle\!\rangle_\rho \triangleq \rho(x)$
$\langle\!\langle b_{A\to B}(a_A)\rangle\!\rangle_\rho \triangleq$
$\quad (\langle\!\langle b\rangle\!\rangle_\rho.arg \Leftarrow \varsigma(x{:}\langle\!\langle A\to B\rangle\!\rangle)\,\langle\!\langle a\rangle\!\rangle_\rho).val \qquad$ for $x\notin FV(\langle\!\langle a\rangle\!\rangle_\rho)$
$\langle\!\langle \lambda(x{:}A)b_B\rangle\!\rangle_\rho \triangleq$
$\quad [arg = \varsigma(x{:}\langle\!\langle A\to B\rangle\!\rangle)\, x.arg,$
$\quad\ val = \varsigma(x{:}\langle\!\langle A\to B\rangle\!\rangle)\,\langle\!\langle b\rangle\!\rangle_{\rho\{x\leftarrow x.arg\}}]$

$A\to B$ is invariant!

Unsoundness of Method Extraction

(Val Extract) (where $A\equiv[l_i{:}B_i\ ^{i\in1..n}]$)

$$\frac{E\vdash a : A \quad j\in 1..n}{E\vdash a\cdot l_j : A\to B_j}$$

$P \triangleq [f{:}[]]$
$Q \triangleq [f{:}[], y{:}[]]$
$Q <: P$

$p : P \triangleq [f=[]]$
$q : Q \triangleq [f=\varsigma(s{:}Q)s.y,\ y=[]]$
then $q : P$ by subsumption with $Q <: P$
hence $q\cdot f : P\to[]$ that is $\lambda(s{:}Q)s.y : P\to[]$

But $q.f(p)$ fails!

Recursive Types

$E \vdash \mu(X)A$	if	$E,X \vdash A$ and $A \succ X$ (A contractive in X)
$E \vdash fold(\mu(X)A, a) : \mu(X)A$	if	$E \vdash a : A\{X \leftarrow \mu(X)A\}$
$E \vdash unfold(a) : A\{X \leftarrow \mu(X)A\}$	if	$E \vdash a : \mu(X)A$

$E \vdash \mu(X)A <: \mu(Y)B$ if $E,Y,X<:Y \vdash A<:B$ and $E \vdash \mu(X)A, E \vdash \mu(Y)B$

In particular:

$\mu(X)[l{:}X, m{:}[], n{:}[]] \not<: \mu(Y)[l{:}Y, m{:}[]]$

Moreover, assuming so would be unsound.

(Note)

This counterexample is a variation of the one against object type covariance. The standard counterexamples for recursive types do not apply to invariant arrows.

$Q = \mu(Y)[l{:}Y, m{:}[]]$ $UQ = [l{:}Q, m{:}[]]$

$P = \mu(X)[l{:}X, m{:}[], n{:}[]]$ $UP = [l{:}P, m{:}[], n{:}[]]$

$q : Q = \mu(x{:}Q)\ fold(Q, [l=x, m=[]])$

$p : P = \mu(x{:}P)\ fold(P, [l=x, m=\varsigma(s{:}UP)unfold(s.l).n, n=[]])$

Then $p : Q$ by subsumption

$(unfold(p).l{:=}q) : Q$ that is $[l=q, m=\varsigma(s{:}UP)unfold(s.l).n, n=[]] : Q$

But $(unfold(p).l{:=}q).m$ fails!

Classes

If $A \equiv [l_i{:}B_i\ ^{i\in 1..n}]$ is an object type, then:

$Class(A) \triangleq [new{:}A, l_i{:}A \rightarrow B_i\ ^{i\in 1..n}]$

Where

$new{:}A$	is a **generator** for objects of type A
$l_i{:}A \rightarrow B_i$	is a **pre-method** for objects of type A

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

We can produce new objects as follows:

$c.new \equiv [l_i{=}\varsigma(x{:}A)b_i\{x\}\ ^{i\in 1..n}] : A$

Inheritance

Define inheritance as a new relation between class types:

$Class(A')$ inh $Class(A)$ iff $A'<:A$

Let $A \equiv [l_i{:}B_i\ ^{i\in 1..n}]$ and $A' \equiv [l_i{:}B_i\ ^{i\in 1..n}, l_j{:}B_j\ ^{j\in n+1..m}]$, with $A' <: A$.

Note that $Class(A') \not<: Class(A)$ and $Class(A) \not<: Class(A')$.

Let $c{:}\ Class(A)$, then

$c.l_i{:}\ A \rightarrow B_i <: A' \rightarrow B_i$.

Hence $c.l_i$ is a good pre-method for $Class(A')$. For example, we may define:

$c' \triangleq [new{=}..., l_i{=}c.l_i\ ^{i\in 1..n}, l_j{=}...\ ^{j\in n+1..m}] : Class(A')$

where class c' **inherits** the methods l_i from class c.

Shortcomings of First-Order Systems

“Expected” first-order subtypings fail, particularly for methods that return self:

P_1	$\triangleq$	$\mu(X)[x{:}Int,\ mv\_x{:}Int\rightarrow X]$	movable 1-D points
P_2	$\triangleq$	$\mu(X)[x,y{:}Int,\ mv\_x,mv\_y{:}Int\rightarrow X]$	movable 2-D points

Unfortunately, $P_2 <: P_1$ is not provable (and inconsistent).

Solutions:

- Avoid methods specialization, to obtain $P_2 <: P_1$ (needs dynamic type tests):

  $P_1 \triangleq \mu(X)[x{:}Int,\ mv\_x{:}Int\rightarrow X]$

  $P_2 \triangleq \mu(X)[x,y{:}Int,\ mv\_x{:}Int\rightarrow P_1,\ mv\_y{:}Int\rightarrow X]$
- Axiomatize primitive “Self” types, such that $P_2 <: P_1$ (axioms are non-trivial):

  $P_1 \triangleq [x{:}Int,\ mv\_x{:}Int\rightarrow Self]$

  $P_2 \triangleq [x,y{:}Int,\ mv\_x,mv\_y{:}Int\rightarrow Self]$
- Use an imperative framework, where the problem is reduced (although not eliminated) by taking mv_x:Unit.
- Move up to the second order, and see what can be done there.

SECOND-ORDER OBJECT CALCULI

Second-Order Calculi

Take a first-order object calculus with subtyping, and add bounded quantifiers:

Bounded universals: (contravariant in the bound)

$E \vdash \forall(X<:A)B$ if $E,X<:A \vdash B$

$E \vdash \forall(X<:A)B <: \forall(X<:A')B'$ if $E \vdash A' <: A$ and $E,X<:A' \vdash B <: B'$

$E \vdash \lambda(X<:A)b : \forall(X<:A)B$ if $E,X<:A \vdash b : B$

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

Bounded existentials: (covariant in the bound)

$E \vdash \exists(X<:A)B$ if $E,X<:A \vdash B$

$E \vdash \exists(X<:A)B <: \exists(X<:A')B'$ if $E \vdash A <: A'$ and $E,X<:A \vdash B <: B'$

$E \vdash (\text{pack } X<:A=C,\ b\{X\}{:}B\{X\}) : \exists(X<:A)B\{X\}$
  if $E \vdash C <: A$ and $E \vdash b\{C\} : B\{C\}$

$E \vdash (\text{open } c \text{ as } X<:A,x{:}B \text{ in } d{:}D) : D$
  if $E \vdash c : \exists(X<:A)B$ and $E \vdash D$ and $E,X<:A,x{:}B \vdash d : D$

Covariant Components

Suppose we have:

Point $\triangleq$ [x,y: Real]
ColorPoint <: Point $\triangleq$ [x,y: Real, c: Color]
Circle $\triangleq$ [center: Point, radius: Real]
ColorCircle $\triangleq$ [center: ColorPoint, radius: Real]

Unfortunately, ColorCircle </: Circle, because of invariance. Now redefine:

Circle $\triangleq \exists$(X<:Point) [center: X, radius: Real]
ColorCircle $\triangleq \exists$(X<:ColorPoint) [center: X, radius: Real]

Thus we gain ColorCircle <: Circle. But covariance in object types was supposed to be unsound, so we must have lost something.

We have lost the ability (roughly) to update the center component, since X is unknown. Therefore covariant components are (roughly) read-only components.

The center component can still be extracted out of the abstraction, by subsumption from X to ColorPoint.

Contravariant Components

There are techniques to obtain contravariant (write-only) components; but these are more complex. (A write-only component can still be read by its sibling methods.) Here is an overview.

$A \triangleq [l{:}B, ...]$ which we want contravariant in B

is transformed into:

$A' \triangleq ... [l_{ovr}{:}Y, l{:}B, ...]$ where $Y{<:}(A'\to B)\to A'$ and l_{ovr} overrides l

A' is still invariant in B, but any element of A' can be subsumed into:

$A'' \triangleq ... [l_{ovr}{:}Y, ...]$ contravariant in B, with $A'{<:}A''$

The appropriate definitions are:

$$A' \triangleq \mu(X)\ \exists(Y{<:}(X\to B)\to X)\ [l_{ovr}{:}Y, l{:}B, ...]$$

$$A'' \triangleq \mu(X)\ \exists(Y{<:}(X\to B)\to X)\ [l_{ovr}{:}Y, ...]$$

Then $o.l \Lleftarrow \varsigma(s{:}A)b$ is simulated by a definable override(o', l_{ovr}, $\lambda(s{:}A'')b''$) (i.e., roughly, $o.l_{ovr}(\lambda(s{:}A)b)$) for appropriate transformations of $o{:}A$ into $o'{:}A'$ and b into b''.

Variant Product and Function Types

Encodings based on object types alone may be undesirably invariant. Quantifiers can introduce the necessary degree of variance.

Variant product types can be define as:

$$A \dot{\times} B \triangleq \exists(X{<:}A)\ \exists(Y{<:}B)\ [fst{:}X, snd{:}Y]$$

With the property:

$A \dot{\times} B <: A' \dot{\times} B'$ if $A <: A'$ and $B <: B'$

Similarly, but somewhat more surprisingly, we can obtain variant function types:

$$A \dot{\to} B \triangleq \forall(X{<:}A)\ \exists(Y{<:}B)\ [arg{:}X, val{:}Y]$$

With the property:

$A \dot{\to} B <: A' \dot{\to} B'$ if $A' <: A$ and $B <: B'$

Translation of the first-order λ-calculus with subtyping:

$$\langle\!\langle A\to B\rangle\!\rangle \triangleq \forall(X{<:}\langle\!\langle A\rangle\!\rangle)\ \exists(Y{<:}\langle\!\langle B\rangle\!\rangle)\ [arg{:}X, val{:}Y]$$

$$\langle\!\langle x_A\rangle\!\rangle_\rho \triangleq \rho(x)$$

$$\begin{aligned}&\langle\!\langle b_{A\to B}(a_A)\rangle\!\rangle_\rho \triangleq\\ &\quad \text{open } \langle\!\langle b\rangle\!\rangle_\rho(\langle\!\langle A\rangle\!\rangle) \text{ as } Y{<:}\langle\!\langle B\rangle\!\rangle, y{:}[arg{:}\langle\!\langle A\rangle\!\rangle, val{:}Y]\\ &\quad \text{in } (y.arg \Lleftarrow \varsigma(x{:}[arg{:}\langle\!\langle A\rangle\!\rangle, val{:}Y])\ \langle\!\langle a\rangle\!\rangle_\rho).val \qquad \text{for } Y,y,x \notin FV(\langle\!\langle a\rangle\!\rangle_\rho)\end{aligned}$$

$$\begin{aligned}&\langle\!\langle \lambda(x{:}A)b_B\rangle\!\rangle_\rho \triangleq\\ &\quad \lambda(X{<:}\langle\!\langle A\rangle\!\rangle)\\ &\qquad (\text{pack } Y{<:}\langle\!\langle B\rangle\!\rangle{=}\langle\!\langle B\rangle\!\rangle,\\ &\qquad\quad [arg = \varsigma(x{:}[arg{:}X, val{:}\langle\!\langle B\rangle\!\rangle])\ x.arg,\\ &\qquad\quad\ val = \varsigma(x{:}[arg{:}X, val{:}\langle\!\langle B\rangle\!\rangle])\ \langle\!\langle b\rangle\!\rangle_{\rho\{x\leftarrow x.arg\}}]\\ &\qquad\ : [arg{:}X, val{:}Y])\end{aligned}$$

Self Types

Recall that $\mu(X)B$ failed to give some expect subtyping behavior. We are now looking for a different quantifier, $\varsigma(X)B$, with the expected behavior.

$P_1 \triangleq \varsigma(Self)[x{:}Int, mv\_x{:}Int\to Self]$		movable 1-D points
$P_2 \triangleq \varsigma(Self)[x,y{:}Int, mv\_x,mv\_y{:}Int\to Self]$		movable 2-D points

Let $P_1(X) \triangleq [x{:}Int, mv\_x{:}Int\to X]$ be the **X-unfolding** of P_1

with $P_1(P_1) \equiv [x{:}Int, mv\_x{:}Int\to P_1]$ the **self-unfolding** of P_1.

Some properties we expect for $\varsigma(X)B$, are:

- Subtyping E.g.: $P_2 <: P_1$
- Creation (folding) E.g.: from $P_1(P_1)$ to P_1
- Selection (unfolding) E.g.: $p_1.mv\_x{:}\ Int\to P_1$
- Update (refolding)

  E.g.: from $p_1{:}P_1$ and a "Self-parametric" method such that

  for all $Y{<:}P_1$ and $x{:}P_1(Y)$ gives $Int\to Y$

  produce a new P_1 with an updated mv_x

The $\varsigma(X)B$ Quantifier

It turns out that Self can be formalized via a general quantifier, i.e., independently of object types. Define:

$$\boxed{\varsigma(X)B \triangleq \mu(Y)\,\exists(X<:Y)\,B \qquad \text{(Y not occurring in B)}}$$

The intuition is the following. Take A<:A' with A≠A':

Want:	[l:A, m:C]	<:	[l:A']	(fails)
Do:	$\exists$(X<:A) [l:X, m:C]	<:	$\exists$(X<:A') [l:X]	(holds)
Want:	μ(Y) [l:Y, m:C]	<:	μ(Y) [l:Y]	(fails)
Do:	μ(Y) $\exists$(X<:Y) [l:X, m:C]	<:	μ(Y) $\exists$(X<:Y) [l:X]	(holds)

This way we can have, e.g. $P_2<:P_1$. We achieve subtyping at the cost of making certain fields covariant and, hence, essentially read-only. This suggests, in particular, that we will have difficulties in overriding methods that return self.

(Note)

$\varsigma(X)B$ satisfies the subtyping property:

$E \vdash \varsigma(X)B <: \varsigma(X)B'$ if $E,X \vdash B <: B'$

even though we do not have, in general, $\mu(X)B <: \mu(X)B'$.

$E,X \vdash B <: B'$

$\Rightarrow$ $E,Z,Y<:Z,X<:Y \vdash B <: B'$ by weakening , for fresh Y,Z

$\Rightarrow$ $E,Z,Y<:Z \vdash \exists(X<:Y)B <: \exists(X<:Z)B'$ by (Sub Exists)

$\Rightarrow$ $E \vdash \mu(Y)\exists(X<:Y)B <: \mu(Z)\exists(X<:Z)B'$ by (Sub Rec) and contractiveness

Building Elements of Type $\varsigma(X)B$

Modulo an unfolding, $\varsigma(X)B \equiv \mu(Y)\exists(X<:Y)B$ (for Y not in B) is the same as:

$\exists(X<:\varsigma(X)B)B$.

An element of $\exists(X<:\varsigma(X)B)B$ is a pair $\langle C, c\rangle$ consisting of a subtype C of $\varsigma(X)B\{X\}$ and an element c of B{C}.

We denote by

$\varsigma\langle C, c\rangle$

the injection of the pair $\langle C, c\rangle$ from $\exists(X<:\varsigma(X)B)B$ into $\varsigma(X)B$.

For example, suppose we have an element x of type $\varsigma(X)X$. Then, choosing $\varsigma(X)X$ as the required subtype of $\varsigma(X)X$, we obtain $\varsigma\langle\varsigma(X)X, x\rangle : \varsigma(X)X$. Therefore we can construct:

$\mu(x)\ \varsigma\langle\varsigma(X)X, x\rangle\ :\ \varsigma(X)X$

The fully explicit version of $\varsigma\langle C, c\rangle$ is written:

$\varsigma(X<:\varsigma(X)B=C)\ c$ (or $\varsigma(X=\varsigma(X)B)\ c$ for $C\equiv\varsigma(X)B$)

and it binds the name X to C in c.

Building a Memory Cell

Suppose we want to build a memory cell m:M with a read operation rd:Nat and a write operation wr:Nat→M. We can define:

$M \triangleq \varsigma(Self)[rd:Nat, wr:Nat\rightarrow Self]$

where the wr method should use its argument to override the rd field. For convenience, we adopt the following abbreviation to unfold a Self quantifier:

$A(C) \triangleq B\{C\}$ whenever $A \equiv \varsigma(X)B\{X\}$ and $C<:A$

For example we have $M(M) \equiv [rd:Nat, wr:Nat\rightarrow M]$.

Then we can define:

```
m: M   ≜   ς⟨M,
              [rd = 0,
               wr = ς(s:M(M)) λ(n:Nat) ς⟨M, s.rd:=n⟩]⟩
```

Derived Rules for $\varsigma(X)B$

Formally, we can define an introduction construct ($\varsigma(Y<:A=C)b\{Y\}$) and an elimination construct (use c as $Y<:A, y:B\{Y\}$ in $d:D$), for $\varsigma(X)B$, such that:

(Type Self)

$$\frac{E,X \vdash B \qquad B \succ X}{E \vdash \varsigma(X)B}$$

(Sub Self)

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

(Val Self) (where $A \equiv \varsigma(X)B\{X\}$)

$$\frac{E \vdash C <: A \qquad E \vdash b\{C\} : B\{C\}}{E \vdash \varsigma(Y<:A=C)b\{Y\} : A}$$

(Val Use) (where $A \equiv \varsigma(X)B\{X\}$)

$$\frac{E \vdash c : A \qquad E \vdash D \qquad E, Y<:A, y:B\{Y\} \vdash d : D}{E \vdash (\text{use } c \text{ as } Y<:A,\ y:B\{Y\} \text{ in } d:D) : D}$$

(Plus the derived equational theory.)

(Note)

Define, for $A \equiv \varsigma(X)B\{X\}$, $C<:A$, and $b\{C\}:B\{C\}$:

$$\varsigma(Y<:A=C)\ b\{Y\} \quad \triangleq \quad \text{fold}(A,\ (\text{pack } Y<:A=C,\ b\{Y\}:B\{Y\}))$$

and, for $c:A$ and $d\{Y,y\}:D$, where Y does not occur in D:

$$(\text{use } c \text{ as } Y<:A,\ y:B\{Y\} \text{ in } d\{Y,y\}:D) \quad \triangleq$$
$$(\text{open unfold}(c) \text{ as } Y<:A,\ y:B\{Y\} \text{ in } d\{Y,y\}:D)$$

The ςOb Calculus

At this point we may extract a **minimal second-order object calculus**. We discard the universal and existential quantifiers, and recursion, and we retain the ς quantifier and the object types:

$$\begin{array}{ll}
A,B ::= & \\
 & X \\
 & Top \\
 & [l_i:B_i\ ^{i\in1..n}] \\
 & \varsigma(X)B
\end{array}
\qquad\qquad
\begin{array}{ll}
a,b ::= & \\
 & x \\
 & [l_i=\varsigma(x_i:A_i)b_i\ ^{i\in1..n}] \\
 & a.l \\
 & a.l \Leftarrow \varsigma(x:A)b \\
 & \varsigma(X<:A=B)b \\
 & \text{use } a \text{ as } X<:A,\ y:B \text{ in } b:D
\end{array}$$

ς-Object Types

Now that we have a general formulation of $\varsigma(X)B$, we can go back and consider its application to object types. We consider types of the special form:

$$\varsigma(X^+)[l_i:B_i\{X\}\ ^{i\in1..n}] \quad \triangleq \quad \varsigma(X)[l_i:B_i\{X\}\ ^{i\in1..n}] \ \text{ when the } B_i \text{ are covariant in } X$$

Here, $\varsigma(X^+)[l_i:B_i\{X\}\ ^{i\in1..n}]$ are called ς-object types. Our goal is to discover their derived typing rules.

- The covariance requirement is necessary to get selection to work. An example of violation of covariance are "binary methods" such as:

  $\varsigma(Self)[\ ...,\ eq: Self \to Bool,\ ...\ ]$

(It turns out that p.eq cannot be given a type, because a contravariant Self occurrence is not able to escape the scope of the existential quantifier. A covariant Self occurrence can be eliminated by subsumption into the object type.)

- The covariance requirements rules out "nested" Self types, because of the invariance of object type components ($\varsigma(Y)\ [l_2: X]$ is invariant in X):

  $\varsigma(X)\ [l_1: \varsigma(Y)\ [l_2: X]]$

- These restrictions are common in languages that admit Self types.

Derived Rules for ς-Object Types

We have essentially the same rules for subtyping and construction. But now, the generic "use" elimination construct of ς-quantifiers can be specialized to obtain selection and override:

(Val ςSelect) (where $A \equiv ς(X^+)[l_i{:}B_i\{X\}^{\ i\in 1..n}]$)

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

(Val ςOverride) (where $A \equiv ς(X^+)[l_i{:}B_i\{X\}^{\ i\in 1..n}]$)

$$\frac{E \vdash a : A \qquad E, Y{<:}A, x{:}A(Y) \vdash b : B_j\{Y^+\} \qquad j\in 1..n}{E \vdash a.l_j \Leftarrow ς(Y{<:}A, x{:}A(Y))b : A}$$

where $ς(Y{<:}A, x{:}A(Y))b$ is a "Self-parametric" method that must produce for every $Y{<:}A$ and $x{:}A(Y)$ (where x is self) a result of type $B_j\{Y^+\}$, parametrically in Y. In particular, it is unsound for the method to produce a result of type $B_j\{A\}$.

Hence the (already known) notion of Self-parametric methods falls out naturally in this framework, as a condition for a derived rule.

(Note)

Assume $a{:}A$ with $A \equiv ς(X^+)[l_i{:}B_i\{X\}^{\ i\in 1..n}]$ and $A(X) \equiv [l_i{:}B_i\{X^+\}^{\ i\in 1..n}]$, and set, with some overloading of notation:

$a.l_j \triangleq$
  (use a as $Z{<:}A$, $y{:}A(Z)$ in $y.l_j : B_i\{A^+\}$)

$a.l_j \Leftarrow ς(Y{<:}A, y{:}A(Y), x{:}A(Y))b\{Y,y,x\} \triangleq$
  (use a as $Z{<:}A$, $y{:}A(Z)$ in $ς(Y{<:}A{=}Z)\ (y.l_j \Leftarrow ς(x{:}A(Y))b\{Y,y,x\}) : A$)

The Type of the Object-Oriented Naturals

We can finally give a type for the object-oriented natural numbers:

$N_{Ob} \triangleq ς(Self^+)[succ{:}Self,\ case{:}\forall(Z)Z\to(Self\to Z)\to Z]$

Note that the covariance restriction is respected.

The zero numeral can then be typed as follows:

```
zero_Ob : N_Ob ≜
  ς(Self=N_Ob)
     [ case = λ(Z) λ(z:Z) λ(f:Self→Z) z,
       succ = ς(n:N_Ob(Self))
                 ς⟨Self, n.case := λ(Z) λ(z:Z) λ(f:Self→Z) f(ς⟨Self,n⟩)⟩]
```

The Type of the Calculator

$C \triangleq ς(Self^+)[arg,acc{:}\ Real,\ enter{:}\ Real\to Self,\ add,sub{:}\ Self,\ equals{:}\ Real]$

$Calc \triangleq ς(Self^+)[enter{:}\ Real\to Self,\ add,sub{:}\ Self,\ equals{:}\ Real]$

Then Calc <: C; we can hide arg and acc from clients.

```
calculator: C ≜
  ς(Self=C)
     [arg = 0.0,
      acc = 0.0,
      enter = ς(s:C(Self)) λ(n:Real) ς⟨Self, s.arg := n⟩,
      add = ς(s:C(Self))
         ς⟨Self, (s.acc := s.equals).equals ⇐ ς(s':C(Self)) s'.acc+s'.arg⟩,
      sub = ς(s:C(Self))
         ς⟨Self, (s.acc := s.equals).equals ⇐ ς(s':C(Self)) s'.acc-s'.arg⟩,
      equals = ς(s:C(Self)) s.arg ]
```

Overriding and Self

If we want to override a method of a ς-object o:A, the new method must work for any possible Self<:A, because o might have been initially built as an element of an unknown B<:A.

This is a tough requirement if the method result involves the Self type, since we do not know the “true Self” of o.

(We have no such problem at object creation time, since the “true Self” is known then. But the same difficulty would likely surface if we were creating objects incrementally, adding one method at a time to extensible objects.)

Consider, for example, the type:

$$A \triangleq \varsigma(Self^{+})[n{:}Int,\ m{:}Self] \qquad \text{with} \qquad A(Self) \equiv [n{:}Int,\ m{:}Self]$$

According to the rule (Val ςOverride), an overriding method can use in its body the variables Self<:A, and x:A(Self), where x is the self of the new method.

Basically, for a method l with result type $B_l\{Self\}$, the override rule requires that we construct a polymorphic function of type:

$$\forall(Self<:A)\ A(Self) \to B_l\{Self\}$$

For n, we have no problem in returning a $B_n\{Self\} \equiv Int$.

But for m, there is no obvious way of producing a $B_m\{Self\} \equiv Self$ from x:A(Self), except for x.m which loops. And we cannot construct an element of an arbitrary Self<:A.

Moreover, using $\forall(Self<:A)\ A(Self) \to B\{A\}$, for example, would be unsound.

In conclusion, the (Val ςOverride) rule, although sufficient for overriding simple methods and fields, is not sufficient to allow us to *usefully* override methods that return a value of type Self, *after object construction.*

Recoup

We introduce a special method called *recoup* with an associated run-time invariant. Recoup is a method that returns self immediately. The invariant asserts that the result of recoup is its host object. These simple assumptions have surprising consequences.

$$A \triangleq \varsigma(Self^{+})[r{:}Self,\ n{:}Int,\ m{:}Self] \qquad \text{with } A(Self) \equiv [r{:}Self,\ n{:}Int,\ m{:}Self]$$

$$a : A \triangleq \varsigma(Self<:A{=}A)\ [r = \varsigma(x{:}A(Self))\varsigma\langle Self,x\rangle,\ ...\ ] : A$$

Then, the following override on m typechecks, since x.r has type Self:

$$a.m \Leftarrow \varsigma(Self<:A,\ x{:}A(Self))\ (x.n{:=}3).r \quad : \quad A$$

The reduction behavior of this term relies on the recoup invariant. I.e., recoup should be correctly initialized and not subsequently corrupted.

Intuitively, recoup allows us to recover a “parametric self ” x.r which equals the object a but has type Self<:A (the “true Self”) and not just type A (the “known Self”).

In general, if A has the form $\varsigma(Self^{+})[r{:}Self,\ ...]$ then we can write *useful* polymorphic functions of type:

$$\forall(Self<:A)\ A(Self) \to Self$$

that are not available without recoup. Such functions are parametric enough to be useful for method override.

In a programming language based on these notions, recoup could be introduced as a “built-in feature”, so that the recoup invariant is guaranteed for all objects at all times.

CONCLUSIONS

- An untyped object calculus with method override.
- First-order typing, with subsumption.
- Second-order typing, with Self types.
- Able to encode various λ-calculi.

- A CUPER semantics, and a coherent translation.
- An imperative version, with typing soundness.

- Object calculi taken to be as “fundamental” as λ-calculi.
- The final typed formal system is a *standard* second-order extension of an *elementary* first-order object calculus. In particular, no higher-order constructs.

REFERENCES

[Abadi, Cardelli 1994a] M. Abadi and L. Cardelli, **An imperative object calculus**. Manuscript.

[Abadi, Cardelli 1994b] M. Abadi and L. Cardelli. **A semantics of object types**. *Proc. IEEE Symposium on Logic in Computer Science (to appear).*

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

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

[Cardelli 1988] L. Cardelli, **A semantics of multiple inheritance**. *Information and Computation* **76**, 138-164.

[Cook 1989] W.R. Cook. **A denotational semantics of inheritance**. Ph.D. Thesis, Computer Science Dept., Brown University.

[Kamin 1988] S. Kamin. **Inheritance in Smalltalk-80: a denotational definition**. *Proc. 15th Annual ACM Symposium on Principles of Programming Languages*.

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