Object Types with Self

Martín Abadi and Luca Cardelli

Digital Equipment Corporation, Systems Research Center

CONTENTS

INTRODUCTION

- We describe a primitive second-order theory of object types with Self.
- As a main novelty, the type rules are based on structural subtyping assumptions within a second-order system. These assumptions fail in common denotational interpretations, but are sound with respect to any natural operational semantics, and lead to great expressiveness.
- As examples of that expressiveness, we can write natural-looking code and types for "moving points", we can override methods that return Self, and we can define classes and inheritance effectively.
- As an application, we provide a close emulation of TOOPLE's type rules.

OBJECT TYPES WITH SELF

An Untyped Object Calculus

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 override	
	$\lambda()b$	type abstraction (type-erased $\lambda(X<:A)b$)	
	b()	type application (type-erased b(A))	

Field Notation: l=b stands for $l=\varsigma(x)b$ with $x \notin b$
a.l:=b stands for $a.l \Leftarrow \varsigma(x)b$ with $x \notin b$

Functions: $\lambda(x)b$, b(a) can be encoded.

Operational Semantics

v ::=		
	$[l_i=\varsigma(x_i)b_i \ ^{i\in 1..n}]$	object result (l_i distinct)
	$\lambda()b$	type abstraction result

$\vdash a \leadsto v$	term reduction judgment (with a fixed evaluation order)

(Red Object)

$$\frac{}{\vdash [l_i=\varsigma(x_i)b_i \ ^{i\in 1..n}] \leadsto [l_i=\varsigma(x_i)b_i \ ^{i\in 1..n}]}$$

(Red Select)

$$\frac{\vdash a \leadsto [l_i=\varsigma(x_i)b_i \ ^{i\in 1..n}] \quad \vdash b_j\{x_j \leftarrow [l_i=\varsigma(x_i)b_i \ ^{i\in 1..n}]\} \leadsto v \quad j\in 1..n}{\vdash a.l_j \leadsto v}$$

(Red Override)

$$\frac{\vdash a \leadsto [l_i=\varsigma(x_i)b_i \ ^{i\in 1..n}] \quad j\in 1..n}{\vdash a.l_j \Leftarrow \varsigma(x)b \leadsto [l_j=\varsigma(x)b, l_i=\varsigma(x_i)b_i \ ^{i\in 1..n-\{j\}}]}$$

(Red Fun2)

$$\frac{}{\vdash \lambda()a \leadsto \lambda()a}$$

(Red Appl2)

$$\frac{\vdash b \leadsto \lambda()a \quad \vdash a \leadsto v}{\vdash b() \leadsto v}$$

Second-Order Typing

A,B ::=		type
	X	type variable
	Top	maximum type
	$Obj(X)[l_i\upsilon_i:B_i\{X\} \ ^{i\in 1..n}]$	object type ($\upsilon_i \in \{^-,^o,^+\}$, l_i distinct)
	$\forall(X<:A)B$	bounded universal type

Ex: Point $\triangleq$ $Obj(Self)[x^o:Nat, y^o:Nat, mv^o:Nat\times Nat\to Self]$

Note: $\times$ and $\to$ can be encoded from Obj. Then Nat, +, and $\exists$ can be encoded from $\forall$ and $\to$.

Note: with μ and $\exists$ we can define $\varsigma(X)B\{X\} \triangleq \mu(Y)\exists(X<:Y)B\{X\}$. The main operative differences between ς and Obj are:

- ς is sound denotationally, in parametric models. But we need to resort to a "recoup" technique to achieve complete expressiveness.
- Obj is sound only operationally (for now) but, unlike ς, it can "move points", override self-returning methods, and encode classes.

Variance Annotations

There are three field variances (υ): invariant (^{o}), covariant ($^+$), contravariant ($^-$). Ignoring the Obj(X) part for now, we have:

$[\ldots\ l^o{:}B\ \ldots] <: [\ldots\ l^o{:}B'\ \ldots]$	if $B \equiv B'$	invariant
$[\ldots\ l^+{:}B\ \ldots] <: [\ldots\ l^+{:}B'\ \ldots]$	if $B <: B'$	covariant (read-only)
$[\ldots\ l^-{:}B\ \ldots] <: [\ldots\ l^-{:}B'\ \ldots]$	if $B' <: B$	contravariant (write-only)
$[\ldots\ l^o{:}B\ \ldots] <: [\ldots\ l^+{:}B'\ \ldots]$	if $B <: B'$	invariant <: covariant
$[\ldots\ l^o{:}B\ \ldots] <: [\ldots\ l^-{:}B'\ \ldots]$	if $B' <: B$	invariant <: contravariant

A "fourth variance" completes the rules:

$[\ldots\ l\upsilon{:}B\ \ldots] <: [\ldots\ \ \ldots]$ existent <: non existent

Notation: $l{:}B$ stands for $l^o{:}B$.

Some Encodings

$[l_i{:}B_i\ ^{i\in 1..n}]$	$\triangleq$	$\mathrm{Obj}(X)[l_i{}^o{:}B_i\ ^{i\in 1..n}]$	$X \notin B_i\ ^{i\in 1..n}$	Simple objects
$\langle l_i{:}B_i\ ^{i\in 1..n}\rangle$	$\triangleq$	$\mathrm{Obj}(X)[l_i{}^+{:}B_i\ ^{i\in 1..n}]$	$X \notin B_i\ ^{i\in 1..n}$	Records
$A \to B$	$\triangleq$	$[arg^-{:}A,\ val^+{:}B]$ (:> $[arg^o{:}A,\ val^o{:}B]$)		Functions
Nat	$\triangleq$	$\mathrm{Obj}(X)[succ^o{:}X,\ case^o{:}\forall(Z{<:}Top)Z \to (X \to Z) \to Z]$		O-O Naturals

ChurchNat, $\times$, $+$, variants, $\exists$, etc. as in $F_{<:}$ (for example).

(Note) Encodings of Variant Function Types

We first define invariant function types, then subsume:

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

$\lambda(x)b\{x\} \triangleq [arg{=}\varsigma(x)x.arg,\ val{=}\varsigma(x)b\{x.arg\}]$

$b(a) \triangleq (b.arg{:=}a).val$

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

Alternatively, using quantifiers instead of contravariant fields:

$A \to B \triangleq \forall(X{<:}A)[arg^o{:}X,\ val^+{:}B]$

$\lambda(x)b\{x\} \triangleq \lambda()[arg{=}\varsigma(x)x.arg,\ val{=}\varsigma(x)b\{x.arg\}]$

$b(a) \triangleq (b().arg := a).val$

The Structural Subtyping Assumption

"*Every subtype of an object type is an object type*."

(A special case of the override rule)

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

Intuitively, any object type $A <: [l_i{:}B_i\ ^{i\in 1..n}]$ must have the form $[l_i{:}B_i\ ^{i\in 1..n+m}]$. This can be proven syntactically easily.

Consider, though, the special case with $A \equiv X$:

(Specializing for type variables)

$$\frac{E \vdash a : X \qquad E \vdash X <: [l_i : B_i \ ^{i \in 1..n}] \qquad E \vdash b : B_j \quad j \in 1..n}{E \vdash a.l_j := b : X}$$

The rule is operationally sound because X, in the course of a computation, is always instantiated to a closed object type $A <:$ (a closure of) $[l_i:B_i \ ^{i \in 1..n}]$.

However, the rule is not sound in the usual interpretations of $F_{<:}$, for <u>any</u> interpretation of object types!

$\lambda()\ \lambda(x)\ x.l:=3$	not an identity.
$:\ \forall(X<:[l:Nat])X \rightarrow X$	a “type of identities only” [Bruce, Longo 1990]

Type Rules

$E \vdash \diamond$	well-formed environment judgment
$E \vdash A$	type judgment
$E \vdash A <: B$	subtyping judgment
$E \vdash a : A$	value typing judgment

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

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

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

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

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

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

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

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

(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) (l_i distinct) (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 $A' \equiv Obj(X)[l_i \upsilon_i : B_i\{X\} \ ^{i \in 1..n}]$)
$$\frac{E \vdash a : A \qquad E \vdash A <: A' \qquad \upsilon_j \in \{^o, ^+\} \qquad j \in 1..n}{E \vdash a.l_j : B_j\{A\}}$$

(Val Override) (where $A' \equiv Obj(X)[l_i \upsilon_i : B_i\{X\} \ ^{i \in 1..n}]$)
$$\frac{E \vdash a : A \qquad E \vdash A <: A' \qquad E, Y<:A, x{:}Y \vdash b : B_j\{Y\} \qquad \upsilon_j \in \{^o, ^-\} \qquad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x) b : A}$$

(Val Fun2)
$$\frac{E, X<:A \vdash b : B}{E \vdash \lambda()b : \forall(X<:A)B}$$

(Val Appl2)
$$\frac{E \vdash a : \forall(X<:A)B\{X\} \qquad E \vdash A' <: A}{E \vdash a() : B\{A'\}}$$

Soundness

Our type system (including the rules based on structural assumptions!) is sound for the operational semantics:

Theorem (Subject reduction)

If $\varnothing \vdash a : C \ \wedge\ \vdash a \leadsto v$

then $\varnothing \vdash v : C$.

(By induction on the derivation of $\vdash a \leadsto v$.)

Note: In the (Val Override) case, for example, the proof shows that at the point where an override is reduced, the "intermediate" type A of the rule is a closed type that is bound between two concrete object types. Hence A always turns out to be a closed object type during evaluation, satisfying the structural subtyping assumption.

Note: It is sufficient to consider empty environments: a "program" always starts execution from an empty environment. In a "more operational" formulation of the operational semantics, using stacks and closures instead of formal substitution, there is a corresponding theorem with an initial environment E and an associated initial stack *S* that matches it.

(Note) Obj vs. μ

Consider primitive object types with Self, $Obj(X)[l_i{:}B_i\{X\}^{\,i\in 1..n}]$, versus $\mu(X)A\{X\}$, where $A\{X\}\equiv[l_i{:}B_i\{X\}^{\,i\in 1..n}]$ are simple primitive object types.

Better at subtyping

$\mu(X)[l_i{:}B_i\{X\}^{\,i\in 1..n+m}] \ \not<:\ \mu(X)[l_i{:}B_i\{X\}^{\,i\in 1..n}]$

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

Worse at isomorphism

$C \equiv Obj(X)[l_i{:}B_i\{X\}^{\,i\in 1..n}] \ \not\approx\ Obj(X)[l_i{:}B_i\{C\}^{\,i\in 1..n}]$ (can't extract methods)

$C' \equiv \mu(X)[l_i{:}B_i\{X\}^{\,i\in 1..n}] \ \approx\ [l_i{:}B_i\{C'\}^{\,i\in 1..n}]$

Same for type unfolding on invocation

$a{:}C$ implies $a.l_j : B_j\{C\}$

$a'{:}C'$ implies $a'.l_j : B_j\{C'\}$

More restrictive on override

Obj requires an overriding method to be "parametric in Self".

μ has no such requirement.

Memory Cells

(A Compact "Movable Points" Example)

$Mem \triangleq Obj(X)[get^{o}{:}Nat,\ set^{o}{:}Nat\to X]$

$m \triangleq [get = 0,\ set = \varsigma(x)\ \lambda(n)\ x.get{:=}n]$

In an explicitly typed version of the calculus, this looks like:

$m \triangleq obj(Self{=}Mem)$
$\quad [get = 0,$
$\quad\ set = \varsigma(self{:}Self)\ \lambda(n{:}Nat)\ self.get{:=}n]$

N.B. the code and typing is "natural". In particular, there are not extensible records, no coercions, no folding/unfolding, no higher-order operators, no special subtyping relations, etc.

N.B. $Obj(X)[get^{o}{:}Nat,\ set^{o}{:}Nat\to X] \ <:\ Obj(X)[get^{+}{:}Nat,\ set^{+}{:}Nat\to X]$

So, a memory cell can be "protected".

$Mem \triangleq Obj(X)[get^{o}{:}Nat,\ set^{o}{:}Nat\to X]$

$m \triangleq [get = \varsigma(x)0,\ set = \varsigma(x)\ \lambda(n)\ x.get{:=}n]$

<u>**Show**</u>: $m : Mem$

$\varnothing, x{:}Mem \vdash 0 : Nat$	Nat intro
$\varnothing, x{:}Mem, n{:}Nat \vdash x : Mem$	(Val x)
$\varnothing, x{:}Mem, n{:}Nat \vdash Mem <: Mem$	(Sub Refl)
$\varnothing, x{:}Mem, n{:}Nat, Y{<:}Mem, z{:}Y \vdash n : Nat$	(Val x)
$\varnothing, x{:}Mem, n{:}Nat \vdash x.get \Leftarrow \varsigma(z)n : Mem$	(Val Override)
$\varnothing, x{:}Mem \vdash \lambda(n)x.get \Leftarrow \varsigma(z)n : Nat\to Mem$	$\to$ intro
$\varnothing \vdash [get{=}\varsigma(x)0,\ set{=}\varsigma(x)\lambda(n)x.get \Leftarrow \varsigma(z)n] : Mem$	(Val Object)

$\mathrm{Mem} \triangleq \mathrm{Obj}(X)[\mathrm{get}^{o}{:}\mathrm{Nat},\ \mathrm{set}^{o}{:}\mathrm{Nat}\to X]$

Show: $\lambda(m)\ m.\mathrm{get}{:=}3\ :\ \mathrm{Mem}\to\mathrm{Mem}$ Easy.

Show: $\lambda()\ \lambda(m)\ m.\mathrm{get}{:=}3\ :\ \forall(X{<:}\mathrm{Mem})X\to X$ Remarkable!

$\emptyset, X{<:}\mathrm{Mem}, m{:}X \vdash m : X$	(Val x)
$\emptyset, X{<:}\mathrm{Mem}, m{:}X \vdash X <: \mathrm{Mem}$	(Sub X)
$\emptyset, X{<:}\mathrm{Mem}, m{:}X, Y{<:}X, x{:}Y \vdash 3 : \mathrm{Nat}$	Nat intro
$\boxed{\emptyset, X{<:}\mathrm{Mem}, m{:}X \vdash m.\mathrm{get}\Leftarrow\varsigma(x)3 : X}$	(Val Override)
$\emptyset, X{<:}\mathrm{Mem} \vdash \lambda(m)\ m.\mathrm{get}\Leftarrow\varsigma(x)3 : X\to X$	$\to$ intro
$\emptyset \vdash \lambda()\ \lambda(m)\ m.\mathrm{get}\Leftarrow\varsigma(x)3 : \forall(X{<:}\mathrm{Mem})X\to X$	(Val Fun2)

The override is "parametric in self" because it updates the current self, and hence preserves whatever additional components self may have (including unknown ones).

$\mathrm{Mem} \triangleq \mathrm{Obj}(X)[\mathrm{get}^{o}{:}\mathrm{Nat},\ \mathrm{set}^{o}{:}\mathrm{Nat}\to X]$

Show: $\lambda(m)\ m.\mathrm{set} \Leftarrow \varsigma(x)\lambda(n)x.\mathrm{get}{:=}0\ :\ \mathrm{Mem}\to\mathrm{Mem}$

Not typable with ς, even though there are no quantifiers!

Show: $\lambda()\ \lambda(m)\ m.\mathrm{set} \Leftarrow \varsigma(x)\lambda(n)x.\mathrm{get}{:=}0\ :\ \forall(X{<:}\mathrm{Mem})X\to X$ Remarkable!

Let $E_0 \equiv \emptyset, X{<:}\mathrm{Mem}, m{:}X$

$E_0 \vdash m : X$	(Val x)
$E_0 \vdash X <: \mathrm{Mem}$	(Sub X)
$E_0, Y{<:}X, x{:}Y, n{:}\mathrm{Nat} \vdash x : Y$	(Val x)
$E_0, Y{<:}X, x{:}Y, n{:}\mathrm{Nat} \vdash Y <: \mathrm{Mem}$	(Sub X, Trans)
$E_0, Y{<:}X, x{:}Y, n{:}\mathrm{Nat}, Z{<:}Y, z{:}Z \vdash 0 : \mathrm{Nat}$	Nat intro
$\boxed{E_0, Y{<:}X, x{:}Y, n{:}\mathrm{Nat} \vdash x.\mathrm{get}\Leftarrow\varsigma(z)0 : Y}$	(Val Override)
$E_0, Y{<:}X, x{:}Y \vdash \lambda(n)x.\mathrm{get}\Leftarrow\varsigma(z)0 : \mathrm{Nat}\to Y$	$\to$ intro **param. in Y!**
$\boxed{E_0 \vdash m.\mathrm{set}\Leftarrow\varsigma(x)\lambda(n)x.\mathrm{get}\Leftarrow\varsigma(z)0 : X}$	(Val Override)
$\emptyset, X{<:}\mathrm{Mem} \vdash \lambda(m)\ m.\mathrm{set}\Leftarrow\varsigma(x)\lambda(n)x.\mathrm{get}\Leftarrow\varsigma(z)0 : X\to X$	$\to$ intro
$\emptyset \vdash \lambda()\ \lambda(m)\ m.\mathrm{set}\Leftarrow\varsigma(x)\lambda(n)x.\mathrm{get}\Leftarrow\varsigma(z)0 : \forall(X{<:}\mathrm{Mem})X\to X$	(Val Fun2)

Pre-Methods

Types of the form

$\forall(X{<:}A)X\to B\{X\}$ (with $B\{X\}$ covariant in X)

are useful for:

- overriding self-returning methods.
- more practically, for defining *classes as collections of pre-methods*.

A *pre-method* is a function that is later used to construct a method. E.g. for objects of type $A \equiv \mathrm{Obj}(X)[l_i \upsilon_i{:}B_i\{X\}\ ^{i\in 1..n}]$, a pre-method for l_j has type:

$f : \forall(X{<:}A)X\to B_j\{X\}$

With it, we can create objects as follows:

$[\ \ldots\ l_j{=}\varsigma(s)f()(s)\ \ldots\ ]\ :\ A$ since $s{:}A \vdash f(_A)(s) : B\{A\}$

Pre-methods support *specialization*. A pre-method must work for a collection of possible subtypes, parametrically in Self, so that it can be inherited and specialized to any of these subtypes. This is precisely what a type of the form $\forall(X{<:}A)X\to B\{X\}$ expresses.

Classes as Collections of Pre-Methods

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

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

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

$c : \mathrm{Class}(A)\ \triangleq\ [\mathrm{new} = \varsigma(c_{\mathrm{Class}(A)})\ [l_i{=}\varsigma(s_A)\ c.l_i(_A)(s)\ ^{i\in 1..n}],$
$l_i{=}\ \ldots\ ^{i\in 1..n}]$

The l_i are filled with pre-methods.

Classes are first-class values; x.new is legal for any x:Class(A).

Note: the pre-methods l_i do not normally use the self of the class, but new does.

Inheritance as Pre-Method Reuse

We can now consider the inheritance relation between classes. Suppose we have another type A’ <: A, with a corresponding class type Class(A’):

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

$$A' \equiv Obj(X)[l_i \upsilon'_i{:}B'_i\{X\}^{\ i\in 1..n+m}] \quad <: \quad A$$
$$Class(A') \equiv [new{:}A',\ l_i{:}\forall(X<:A')X\rightarrow B'_i\{X\}^{\ i\in 1..n+m}]$$

Inheritance works as follows.

(1) If c:Class(A) then $c.l_i : \forall(X<:A)X\rightarrow B_i\{X\}$ is a pre-method for Class(A).

(2) Further, if we happen to have $\forall(X<:A)X\rightarrow B_i\{X\} <: \forall(X<:A')X\rightarrow B_i'\{X\}$ then $c.l_i : \forall(X<:A')X\rightarrow B_i'\{X\}$ by subsumption.

(3) Then $c.l_i$ is a legal pre-method for Class(A’). Hence $c.l_i$ can be reused to build classes of type Class(A’): it can be inherited.

The inclusion in (2) holds in virtually all cases of interest. In particular, it holds if A’ is an extension of A (i.e. $B_i\{X\}\equiv B'_i\{X\}\ \forall i\in 1..n$), as is normally the case when building subclasses by extension of superclasses. However, in general inheritability, must be considered carefully.

Subclasses and Inheritability of Pre-Methods

We define (under the previous A,A’, with A’<:A):

Class(A’) *subclass of* Class(A) iff $\forall i\in 1..n.\ \mathit{inheritable}_{A,A'}(l_i)$
$\mathit{inheritable}_{A,A'}(l_i)$ iff $X<:A' \Rightarrow B_i\{X\}<:B_i'\{X\}$

When l_i is inheritable, then the inclusion in (2) holds easily. Now, when does inheritability hold? Lets us consider just the cases with $\upsilon_i\equiv\upsilon_i'$:

- For invariant components, the inheritability of l_i is guaranteed since in this case $B_i\{X\}\equiv B'_i\{X\}$.
- For contravariant components, by a lemma we have that X<:A’ always implies B{X}<:B’{X}; inheritability is guaranteed.
- For covariant components that are properly included, we do not have inheritability. (We do, of course, if $B_i\{X\}\equiv B_i'\{X\}$, which is often the case.)

  Counterexample: if $A' \equiv [l^+{:}Nat]$ and $A \equiv [l^+{:}Int]$, and c : [new:A, l:∀(X<:A)X→Int], then c.l cannot be inherited into Class(A’) ≡ [new:A’, l:∀(X<:A’)X→Nat], because it would produce a bad result.

In conclusion, covariant subtyping induces mild (but essential) restrictions in subclassing. Invariant and contravariant subtyping induce no restrictions.

Interlude: Explicitly-Typed Terms

For simplicity, we have so far worked with untyped terms. However, there is a version of the object calculus where terms have type annotations.

a,b ::=	term
x	variable
$obj(X{=}A)[l_i{=}\varsigma(x_i{:}X)b_i^{\ i\in 1..n}]$	object (l_i distinct)
$a.l$	method invocation
$a.l \Lleftarrow \varsigma(Y<:A,x{:}Y)b$	method override
$\lambda(X<:A)b$	type abstraction
$b(A)$	type application

The rules for type formation and subtyping are identical. The typing rules for terms and the operational semantics are adapted in a straightforward way. The soundness theorem is correspondingly adapted.

a,b ::=	term
x	variable
$obj(X{=}A)[l_i{=}\varsigma(x_i{:}X)b_i^{\ i\in 1..n}]$	object (l_i distinct)
$a.l$	method invocation
$a.l \Lleftarrow \varsigma(Y<:A,x{:}Y)b$	method override

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

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

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

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

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

$$\frac{E\vdash a : A \qquad E\vdash A<:A' \qquad E, Y<:A, x{:}Y \vdash b\{Y,x\} : B_j\{Y\} \qquad \upsilon_j\in\{^o,^-\} \qquad j\in 1..n}{E\vdash a.l_j \Lleftarrow \varsigma(Y<:A, x{:}Y)b\{Y,x\} : A}$$

Something like TOOPLE

We use the idea of classes as collection of pre-methods to "emulate" the syntax and type rules of TOOPLE. Plus polymorphism and prototypes.

A,B ::=	*Bonus*
X, $A \to B$	$\forall(X{<:}A)B$
$Object(X)[l_i \upsilon_i{:}B_i\{X^+\}^{\,i \in I}]$	variance annotations
$Class(X)[l_i \upsilon_i{:}B_i\{X^+\}^{\,i \in I}]$	
a,b ::=	*Bonus*
x, $\lambda(x{:}A)b$, $b(a)$	$\lambda(X{<:}A)B$, $b(A)$
class $(x{:}X{<:}A)\ l_i{=}b_i\ ^{i \in I}$ end	
extend a with $(x{:}X{<:}A)\ l_i{=}b_i\ ^{i \in I}$ end	
override a by $(x{:}X{<:}A)\ l_i{=}b_i\ ^{i \in I}$ end	
new(a)	object $(x{:}X{=}A)\ l_i{=}b_i\ ^{i \in I}$ end
a.l	
a gets $[l_i{=}b_i\ ^{i \in I}]$	modify a by $(x{:}X{<:}A)\ l_i{=}b_i\ ^{i \in I}$ end

Class-based $\leftarrow$ **BOTH** $\rightarrow$ **Prototype-based**

Object Types

$$Object(X)[l_i \upsilon_i{:}B_i\{X^+\}^{\,i \in 1..n}]$$

$$\triangleq\ Obj(X)[l_i \upsilon_i{:}B_i\{X\}^{\,i \in 1..n}] \qquad \text{(Identically, for now)}$$

Self must occur only covariantly (i.e., no binary methods, but see later).

For a "normal" o-o language, we may use $\upsilon \equiv ^o$ for value fields (updatable, non-specializable) and $\upsilon \equiv ^+$ for method fields (invoke-only, specializable).

Still, read-only (but specializable) value fields, and overridable (but non-specializable) method fields are supported just as well.

Not much use for $\upsilon \equiv ^-$, except for theoretical encodings.

Derived Rule for Object Types (trivially)

(Type Object) (l_i distinct)

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

Class Types

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

$$Class(X)[l_i \upsilon_i{:}B_i\{X^+\}^{\,i \in 1..n}]$$

$$\triangleq\ [new^+{:}A,\ l_i^{\,+}{:}\forall(X{<:}A)X \to B_i\{X\}^{\,i \in 1..n}]$$

No subtyping relation on class types: A occurs co- and contravariantly.

Both new and l_i are covariant (invoke-only), for simplicity.

To make some methods non-inheritable, hide them by subsumption in the class type, but keep them visible in the object type. Classes with no inheritable methods (of the form $[new^+{:}A]$) enjoy covariant subtyping.

Derived Rule for Class Types (almost trivially)

(Type Class) (l_i distinct)

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

Classes

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

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

$create_A(c) \equiv obj(X{=}A)[l_i{=}\varsigma(s{:}X)\ c.l_i(X)(s)^{\,i \in 1..n}]$

$$class(x{:}X{<:}A)\ l_i{=}b_i\{X,x\}^{\,i \in 1..n}\ end$$

$$\triangleq\ obj(Y{=}C)\ [new = \varsigma(c{:}Y)\ create_A(c),\ l_i = \lambda(X{<:}A)\ \lambda(x{:}X)\ b_i\{X,x\}^{\,i \in 1..n}]$$

A class is a repository of pre-methods, l_i, with a method to generate objects, new.

The code for new is uniform: $create_A(c)$ fetches all the pre-methods of c and packages them into an object by applying them to the object's self.

Derived Rule for Classes

$$\frac{E, X{<:}A, x{:}X \vdash b_i\{X,x\} : B_i\{X\} \quad \forall i \in 1..n}{E \vdash class(x{:}X{<:}A)\ l_i{=}b_i\{X,x\}\ end\ :\ C}$$

Derivation of the Rule for Classes

$$\left\lvert\begin{array}{ll} \quad\left\lceil\, E, c{:}C, s{:}A \vdash c.l_i(A)(s) : B_i\{A\}\right. & \text{Easy} \\ E, c{:}C \vdash obj(X{=}A)[l_i{=}\varsigma(s{:}X)\, c.l_i(X)(s)\ ^{i\in 1..n}] : A & \text{(Val Object)} \\ \quad\left\lceil\, E, c{:}C, X{<:}A, x{:}X \vdash b_i\{X,x\} : B_i\{X\}\right. & \text{(Hyp., Weaken)} \\ E, c{:}C \vdash \lambda(X{<:}A)\, \lambda(x{:}X)b_i\{X,x\} : \forall(X{<:}A)X \to B_i\{X\} & \text{Easy} \end{array}\right.$$

$$\begin{array}{ll} E \vdash obj(Y{=}C)[new{=}\varsigma(c{:}Y)create_A(c),\ \ l_i{=}\lambda(X{<:}A)\lambda(x{:}X)b_i\{X,x\}\ ^{i\in 1..n}] & \\ \quad : [new^+{:}A, l_i^+{:}\forall(X{<:}A)X \to B_i\{X\}\ ^{i\in 1..n}] \equiv C & \text{(Val Object)} \end{array}$$

Subclasses by Class Extension

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

$A' \equiv Object(X)[l_i\upsilon_i{:}B_i\{X\}\ ^{i\in 1..n+m}] <: A$

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

$C' \equiv Class(X)[l_i\upsilon_i{:}B_i\{X\}\ ^{i\in 1..n+m}]$ subclassing from C.

$\boxed{extend\ a\ with(x{:}X{<:}A')\ l_i{=}b_i\{X,x\}\ ^{i\in n+1..n+m}\ end}$

$\triangleq\ [new = \varsigma(c{:}C')\ create_{A'}(c),$
$\quad l_i{=}a.l_i\ ^{i\in 1..n},\quad l_i{=}\lambda(X{<:}A')\lambda(x{:}X)b_i\{X,x\}\ ^{i\in n+1..n+m}]$

Derived Rule for Class Extension

$$\boxed{\dfrac{E \vdash a : C \qquad E, X{<:}A', x{:}X \vdash b_i\{X,x\} : B_i\{X\} \quad \forall i\in n+1..n+m}{E \vdash extend\ a\ with(x{:}X{<:}A')\ l_i{=}b_i\{X,x\}\ ^{i\in n+1..n+m}\ end : C'}}$$

Derivation of the Rule for Class Extension

$$\left\lvert\begin{array}{ll} \quad\left\lceil\, E, c{:}C', s{:}A' \vdash c.l_i(A')(s) : B_i\{A'\} \quad \forall i\in 1..n{+}m\right. & \text{Easy} \\ E, c{:}C' \vdash obj(X'{=}A')[l_i{=}\varsigma(s{:}X')\, c.l_i(X')(s)\ ^{i\in 1..n+m}] : A' & \text{(Val Object)} \\ \quad\left\lceil\, E, c{:}C' \vdash a.l_i : \forall(X{<:}A)X \to B_i\{X\} \quad \forall i\in 1..n\right. & \text{(Val Select)} \\ E, c{:}C' \vdash a.l_i : \forall(X{<:}A')X \to B_i\{X\} \quad \forall i\in 1..n & \text{(Val Subsum.)} \\ \quad\left\lceil\, E, c{:}C', X{<:}A', x{:}X \vdash b_i\{X,x\} : B_i\{X\} \quad \forall i\in n{+}1..n{+}m\right. & \text{(Hyp., Weaken)} \\ E, c{:}C' \vdash \lambda(X{<:}A')\lambda(x{:}X)b_i\{X,x\} : \forall(X{<:}A')X \to B_i\{X\} & \text{Easy} \end{array}\right.$$

$$\begin{array}{ll} E \vdash obj(Y'{=}C')[new = \varsigma(c{:}Y')\ [l_i{=}\varsigma(s{:}A')\ c.l_i(A')(s)\ ^{i\in 1..n+m}], & \\ \qquad l_i{=}a.l_i\ ^{i\in 1..n}, \qquad l_i{=}\lambda(X{<:}A')\lambda(x{:}X)b_j\{X,x\}\ ^{i\in n+1..n+m}] & \\ \quad : [new^+{:}A', l_i^+{:}\forall(X{<:}A')X \to B_i\{X\}\ ^{i\in 1..n+m}] \equiv C' & \text{(Val Object)} \end{array}$$

Subclasses by Class Overriding

Let $J \subseteq 1..n$, $K=(1..n)-J$, where J are the overridden indices,

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

$A' \equiv Object(X)[l_i\upsilon_i{:}B_i\{X\}\ ^{i\in K}, l_i\upsilon_i'{:}B_i'\{X\}\ ^{i\in J}] <: A$

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

$C' \equiv Class(X)[l_i\upsilon_i{:}B_i\{X\}\ ^{i\in K}, l_i\upsilon_i'{:}B_i'\{X\}\ ^{i\in J}]$ subclassing from C

$\boxed{override\ a\ by(x{:}X{<:}A')\ l_i{=}b_i\{X,x\}\ ^{i\in J}\ end}$

$\triangleq\ [new = \varsigma(c{:}C')\ create_{A'}(c),$
$\quad l_i{=}a.l_i\ ^{i\in K},\ \ l_i{=}\lambda(X{<:}A')\ \lambda(x{:}X)\ b_i\{X,x\}\ ^{i\in J}]$

Derived Rule for Class Overriding

$$\boxed{\dfrac{E \vdash a : C \qquad E \vdash A' <: A \qquad E, X{<:}A', x{:}X \vdash b_i\{X,x\} : B_i'\{X\} \qquad \forall i\in J}{E \vdash override\ a\ by(x{:}X{<:}A')\ l_i{=}b_i\{X,x\}\ ^{i\in J}\ end : C'}}$$

Note: we <u>can</u> specialize method types on overriding, since:

$A'{<:}A$ allows $\upsilon_i'B_i' <: \upsilon_iB_i\ ^{i\in J}$ E.g. $\upsilon_i'{\equiv}\upsilon_i{\equiv}^+$ allows $B_i'{<:}B_i$ (proper).

But we <u>cannot</u> inherit methods with a specialized type: the $B_i\ ^{i\in K}$ do not change. C.f. the inheritability condition.

Derivation of the Rule for Class Overriding

$$
\begin{array}{lll}
E, c{:}C', s{:}A' \vdash c.l_i(A')(s) : B_i\{A'\} & \forall i \in K & \text{Easy} \\
E, c{:}C', s{:}A' \vdash c.l_i(A')(s) : B_i'\{A'\} & \forall i \in J & \text{Easy} \\
E, c{:}C' \vdash obj(X'{=}A')[l_i{=}\varsigma(s{:}X')\, c.l_i(X')(s)^{\,i \in 1..n}] : A' & & \text{(Val Object)} \\
E, c{:}C' \vdash a.l_i : \forall(X{<:}A)X{\to}B_i\{X\} & \forall i \in K & \text{(Val Select)} \\
E, c{:}C' \vdash a.l_i : \forall(X{<:}A')X{\to}B_i\{X\} & \forall i \in K & \text{(Val Subsum.)} \\
E, c{:}C', X{<:}A', x{:}X \vdash b_i\{X,x\} : B_i\{X\} & \forall i \in J & \text{(Hyp., Weaken)} \\
E, c{:}C' \vdash \lambda(X{<:}A')\lambda(x{:}X)b_i\{X,x\} : \forall(X{<:}A')X{\to}B_i\{X\} & & \text{Easy}
\end{array}
$$

$$
\begin{array}{ll}
E \vdash obj(Y'{=}C')[new = \varsigma(c{:}Y')\, [l_i{=}\varsigma(s{:}A')\, c.l_i(A')(s)^{\,i \in 1..n+m}], & \\
\qquad l_i{=}a.l_i^{\,i \in K},\ \ l_i{=}\lambda(X{<:}A')\lambda(x{:}X)b_i\{X,x\}^{\,i \in J}] & \\
\qquad : [new^+{:}A', l_i^+{:}\forall(X{<:}A')X{\to}B_i\{X\}^{\,i \in 1..n+m}] \equiv C' & \text{(Val Object)}
\end{array}
$$

For override, we have to check that $a.l_i : \forall(X<:A')X\to B_i\{X\}$ for $i\in K$. We can obtain this by subsumption with $\forall(X<:A)X\to B_i\{X\} <: \forall(X<:A')X\to B_i\{X\}$. The bounds are included, since $A'<:A$. The bodies are identical. So the assumption $A'<:A$ is needed to make sure that the non-overridden methods still work.

Moreover, $A'<A$ enforces, for $i\in J$, $B_i'<:B_i$ for overridden covariant fields, etc. This constrains the result types of the new methods.

Note that there is no subtyping relation between the original class type and the overridden class type.

Object from Classes

new(a)
$\triangleq$ a.new

Note that c can be a variable: classes can be passed around as values as long as we know their class type.

Subsumption can be applied to classes (to hide pre-methods so that they cannot be inherited) and to objects (so they can be reused in less demanding contexts).

Derived Rule for Object Creation

$$\frac{E \vdash a : Class(X)[l_i \upsilon_i{:}B_i^{\,i \in 1..n}]}{E \vdash new(a) : Object(X)[l_i \upsilon_i{:}B_i^{\,i \in 1..n}]}$$

Object Subsumption (of course)

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

Objects (not in TOOPLE, but used in its operational semantics)

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

$object(x{:}X{=}A)\ l_i{=}b_i\{X,x\}^{\,i \in 1..n}$ end $\triangleq$
$obj(X{=}A)\ [l_i = \varsigma(x{:}X)\ b_i\{X,x\}^{\,i \in 1..n}]$

Derived Rule for Objects (TOOPLE-style)

$$\frac{E, X{<:}A, x{:}X \vdash b_i\{X,x\} : B_i\{X\} \qquad \forall i \in 1..n}{E \vdash object(x{:}X{=}A)\ l_i{=}b_i\{X,x\}\ \text{end} : A}$$

A Stronger Derived Rule for Objects

$$\frac{E, x{:}A \vdash b_i\{A,x\} : B_i\{A\} \qquad \forall i \in 1..n}{E \vdash object(x{:}X{=}A)\ l_i{=}b_i\{X,x\}\ \text{end} : A}$$

Building objects is easier than building classes!

Method Invocation

a.l $\triangleq$
a.l

Derived Rule for Method Invocation

$$\frac{E \vdash a : A \qquad E \vdash A <: Object(X)[l_i \upsilon_i{:}B_i\{X\}^{\,i \in 1..n}]}{E \vdash a.l_j : B_j\{A\}}$$

Object Modification (Update/Override) (Update only in TOOPLE)

Let $J \subseteq 1..n$,

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

$\upsilon_i \in \{^{o}, ^{-}\}\ \forall i \in J$

$$\boxed{\text{modify } a \text{ by}(x{:}X{<:}A)\ l_i{=}b_i\{X,x\}^{\ i \in J} \text{ end}}$$

$$\triangleq \quad a(.l_i \Leftarrow \varsigma(X{<:}A, x{:}X) b_i\{X,x\}^{\ i \in J}) \qquad \text{(a sequence of overrides)}$$

$$\text{a gets } [l_i{=}b_i^{\ i \in J}]$$

$$\triangleq \quad \text{modify } a \text{ by}(x{:}X{<:}A)\ l_i{=}b_i^{\ i \in J} \text{ end} \qquad x,X \text{ fresh}$$

Derived Rule for Object Modification

$$\frac{E \vdash a : A \qquad E \vdash A <: A' \qquad E, X{<:}A, x{:}X \vdash b_i\{X,x\} : B_i\{X\} \qquad \forall i \in J}{E \vdash \text{modify } a \text{ by}(x{:}X{<:}A)\ l_i{=}b_i\{X,x\}^{\ i \in J} \text{ end} : A}$$

Comparisons with TOOPLE/TOIL

- Our rules and TOOPLE's rules are rather different in presentation. TOOPLE has the <: and <# subtyping relations. We have a single <: relation with structural subtyping and unrestricted subsumption. Thus we need just a single quantifier for polymorphism. We have a single rule per construct.
- Still, the TOOPLE primitive rules and our derived rules end up typing virtually the same set of programs. We feel that the differences are much more in presentation than in intent or effect.

- We prove soundness of a small kernel language (only 5 cases in the proof!). We can then obtain many TOOPLE-like variations as derived systems, without much effort. E.g. we automatically get TOOPLE plus polymorphism and delegation.
- We can adopt an imperative semantics, instead of a functional one, as shown elsewhere. We have proven soundness of our typing rules (including polymorphism) for that semantics. By the same encoding of classes, we obtain Something like PolyTOIL.

Inheritable Binary Methods

The main, so far untreated, difference with TOOPLE is that it admits contravariant occurrence of Self, allowing, e.g., for limited binary methods.

- The <# relation works for object types with binary methods, for which there is no useful <: relation.
- Once an object with binary methods is created, it can be used but <u>cannot be subsumed</u>.
- However, the <# relation allows binary methods to be inherited in subclasses.

N.B. true multi-methods can be incorporated, e.g. as suggested by Castagna [G.Castagna 1994], but must rely on some form of run-time typechecking.

Comparisons with Pierce-Turner

- Pierce and Turner [Pierce, Turner 1994] propose an encoding of object types of the form:

$$\boxed{PiTu \quad \triangleq \quad \exists(X)\ X \times (X \rightarrow \langle l_i{:}B_i\{X\}^{\ i \in 1..n}\rangle)}$$

- Our rules for $Obj(X)[l_i{:}B_i\{X\}^{\ i \in 1..n}]$ were inspired by an encoding [Abadi, Cardelli 1994a] based on first-order object types, of the form:

$$\boxed{AbCa \quad \triangleq \quad \mu(Y)\ \exists(X{<:}Y)\ [l_i{:}B_i\{X\}^{\ i \in 1..n}]}$$

- A Dec 1988 email message by Luca Cardelli to John Mitchell "Methods have bounded existential type" discusses a Quest program implementing the encoding $\mu(Y)\ B_1 \times \langle l_i{:}\exists(X{<:}Y)X{\times}(X{\rightarrow}B_i\{X\}^{\ i \in 2..n}\rangle)$, which can be written as:

$$\boxed{Ca \quad \triangleq \quad \mu(Y)\ \exists(X{<:}Y)\ X \times (X \rightarrow \langle l_i{:}B_i\{X\}^{\ i \in 1..n}\rangle)}$$

Elements of this type are constructed as:

$$a : Ca \triangleq \mu(self{:}Ca)\ \text{pack } X{<:}Ca{=}Ca \text{ with } \langle self, \lambda(x{:}X)\ \langle l_i{=}b_i\{X,x\}^{\ i \in 1..n}\rangle\ \rangle$$

Are these three encodings (all supporting subtyping) at all related?

We may see the Ca type as an analogue of the PiTu type where the "state" is the entire object, including the methods. This explains the presence of $\mu(Y)\exists(X<:Y)$, telling us that the representation is a subtype of the entire object (in fact, it is exactly the entire object in most cases).

How is the bounded quantifier used? In PiTu there is a difficulty with method invocation: the result type obtained inside the abstraction is $B_i\{X\}$; we must convert $B_i\{X\}$ to $B_i\{PiTu\}$ to exit the abstraction. Pierce and Turner solve this problem by using functorial strength to coerce between those types.

The solution in Ca is much simpler. Since $X<:Ca$ is the given bound and $B_i\{X\}$ is monotonic in X, we have $B_i\{X\} <: B_i\{Ca\}$. Hence we can exit the abstraction just by subsumption; no repackaging is needed.

The Ca encoding was not considered further because, as pointed out in the 1988 correspondence, it has a serious flaw. Since objects are defined recursively, update cannot work properly: the methods are bound to self, and cannot be properly rebound to a different self because of the abstraction.

PiTu does not have this problem because the state is decoupled from the methods. Hence objects need not be defined recursively, and the representation type does not itself contain a troublesome abstraction.

The update problem in Ca was solved via the first-order object types of [Abadi, Cardelli 1994b]. There, objects are not defined recursively, have self built-in, and support update. Hence an object type:

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

can be used to replace the structure:

$$X \times (X\rightarrow\langle l_i:B_i\{X\}\ ^{i\in 1..n}\rangle)$$

This way, we obtain exactly the AbCa type from the Ca type.

The AbCa type still avoids the need for functorial strength.

As a curiosity, we can use the AbCa type as the representation type of a PiTu encoding:

$$\text{pack } X=AbCa \text{ with } \langle [l_i=\varsigma(x_i)b_i\ ^{i\in 1..n}],\quad \lambda(x:X)\,\langle l_i=x.l_i\rangle\rangle$$
$$:\ \exists(X)\ X \times (X\rightarrow\langle l_i:B_i\{X\}\ ^{i\in 1..n}\rangle)$$

obtaining a PiTu implementation where the state is the entire object.

However, again, we need to apply the functorial strength on selection (I do not know that this exists in our type system).

CONCLUSIONS

- We offer a striking example of how operational semantics can justify strong notions of subtyping.
- Structural subtyping assumptions have been used before, but only in first-order contexts [Bruce 1993; Cardelli, Mitchell 1994]. These assumptions do not seem important for ordinary types (although, in retrospect, we can take advantage of them), but are crucial for object types.

- We give a small and fully adequate second-order theory of object types. We can express object types, class types, and method specialization, in the spirit of [Mitchell 1990; Bruce 1993], and we analyze inheritability. We support polymorphism, but we avoid the complications of higher-order typing and row variables.
- We cover both class-based and delegation-based frameworks. Delegation-based frameworks are essntially built-in. We demonstrate the expressibility of class-based frameworks by closely emulating TOOPLE.

REFERENCES

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

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

[Bruce 1993] K. Bruce. **A paradigmatic object-oriented programming language: design, static typing, and semantics**. Technical Report No. CS-92-01, revised. Williams College. To appear in the Journal of Functional Programming.

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

[Cardelli, Mitchell 1994] L. Cardelli and J.C. Mitchell, **Operations on records**. In *Theoretical Aspects of Object-Oriented Programming,* C.A. Gunter and J.C. Mitchell, ed. MIT Press. 295-350.

[G.Castagna 1994] G.Castagna. **Covariance and contravariance: conflict without a cause**. LIENS-DMI. To appear.

[Mitchell 1990] J.C. Mitchell. **Toward a typed foundation for method specialization and inheritance**. *Proc. 17th 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).