A Theory of Primitive Objects
Untyped and First-Order Systems

Martín Abadi and Luca Cardelli

Digital Equipment Corporation, Systems Research Center

Abstract

We introduce simple object calculi that support method override and object subsumption. We give an untyped calculus, typing rules, and equational rules. We illustrate the expressiveness of our calculi and the pitfalls that we avoid.

1. Introduction

Typed λ-calculi have provided a rich foundation for procedural languages, but attempts to use these calculi to model object-oriented languages have not been completely successful. We aim to study the intrinsic properties of objects by developing object calculi that are as simple and fruitful as λ-calculi. Instead of struggling with complex encodings of objects as λ-terms, we take objects as primitive and concentrate on the rules that they should obey.

We investigate calculi that support *method override* in presence of *object subsumption*. Subsumption is the ability to emulate an object by means of another object that has more refined methods. Override is the operation that modifies the behavior of an object, or class, by replacing one of its methods; the other methods are *inherited*.

All common object-oriented languages allow some combination of subsumption and override, and most handle it correctly. However, type correctness is often achieved via rather subtle conditions. We hope to illuminate the origin of some of these conditions, and illustrate the pitfalls that they avoid. At the semantic and type-theoretic level, where one aims for generality, the combination of subsumption and override has proven hard to model; see section 5 for a discussion of previous work. We provide simple calculi that support those features.

We begin this paper with a challenge: finding an adequate type system for an untyped object calculus. We think the challenge is interesting because, first of all, the calculus is patently object-oriented: it has built-in objects, methods with self, and the characteristic semantics of method invocation and override. Second, the calculus is very simple, with just four syntactic forms, and even without functions. Finally, the calculus is expressive: it can encode the untyped λ-calculus, and can express object-flavored examples in a direct way. After describing the untyped calculus, we define first-order type systems and equational theories.

1.1 Primitive Semantics of Objects

We start by investigating the operational semantics of an untyped object-oriented calculus, while keeping in mind some future typing requirements. Our goal in this section is to define a direct semantics of objects, considering them as primitive.

We consider a kernel calculus including *object formation*, *method invocation*, and *method override*. A *method* is a function having a special parameter, often called *self*, that denotes the same object the method belongs to. A *field* is a degenerate method that does not make use of its

self parameter; we talk about *field selection* and *field update*. We use the terms selection and invocation and the terms update and override somewhat interchangeably. Note that it is sometimes desirable to override a field with a proper method, transparently converting passive data into active computation.

To avoid premature commitments, we avoid any explicit encoding of objects in terms of λ-abstraction and application. We describe method invocation directly by substitution. We believe that the semantics of objects given below is natural and suggestive of common implementation techniques.

Primitive semantics

Let $o \equiv [l_i = \varsigma(x_i)b_i^{\ i\in 1..n}]$	(l_i distinct)		
o	is an object with method names l_i and methods $\varsigma(x_i)b_i$		
$o.l_j$	$\twoheadrightarrow$	$b_j\{x_j \leftarrow o\}$	($j \in 1..n$) selection / invocation
$o.l_j \Leftarrow \varsigma(y)b$	$\twoheadrightarrow$	$[l_j = \varsigma(y)b, l_i = \varsigma(x_i)b_i^{\ i\in(1..n)-\{j\}}]$	($j \in 1..n$) update / override

Notation We use indexed notation of the form $\Phi_i^{\ i\in 1..n}$ to denote sequences $\Phi_1,...,\Phi_n$. The symbol "$\twoheadrightarrow$" means "rewrites to"; we use it informally for now. We write $b\{x\}$ to highlight that x may occur free in b. The substitution of a term c for the free occurrences of x in b is written $b\{x \leftarrow c\}$, or $b\{c\}$ where x is clear from context. We use "$\triangleq$" for "equal by definition", "$\equiv$" for "syntactically identical", and "=" for "provably equal" when applied to two terms.

An object is a collection of components $l_i = a_i$, for distinct labels l_i and associated methods a_i; the order of these components does not matter. The object containing a given method is called the method's *host* object. The letter ς (sigma) is used as a binder for the self parameter of a method; $\varsigma(x)b$ is a method with self parameter x, to be bound to the host object, and body b.

A method invocation is written $o.l_j$, where l_j is a label of o. It reduces to the result of the substitution of the host object for the self parameter in the body of the method named l_j.

A method override is written $o.l_j \Leftarrow \varsigma(y)b$. The intent is to replace the method named l_j of o with $\varsigma(y)b$; this is a single operation that involves a construction binding y in b. A method override reduces to a copy of the host object where the overridden method has been replaced by the overriding one. The semantics of override is functional; an override on an object produces a modified copy of the object. In order to make the formal treatment easier we avoid investigating an imperative operational semantics. However, the type theory we develop later is sound even for an imperative interpretation of field update and method override [4].

Self-substitution is at the core of the primitive semantics. Because of this, it is easy to define non-terminating computations without explicit use of recursion. More interestingly, it is possible for a method to return or modify self.

let	$o \triangleq [l = \varsigma(x)x.l]$	then	$o.l \twoheadrightarrow x.l\{x \leftarrow o\} \equiv o.l \twoheadrightarrow ...$
let	$o' \triangleq [l = \varsigma(x)x]$	then	$o'.l \twoheadrightarrow x\{x \leftarrow o'\} \equiv o'$
let	$o'' \triangleq [l = \varsigma(y)\,(y.l \Leftarrow \varsigma(x)x)]$	then	$o''.l \twoheadrightarrow (o''.l \Leftarrow \varsigma(x)x) \twoheadrightarrow o'$

We place particular emphasis on the ability to modify self, as illustrated by this last example. In object-oriented languages, it is very common for a method to modify components of self, although these components are normally value fields and not other methods. Generalizing, we allow methods to override other methods of self, or even themselves. This feature does not significantly complicate the problems that we address. Method override is exploited in rather interesting examples that seem difficult to emulate in other calculi.

We should stress that our choice of primitives is not without alternatives.

For example, we could have tried to provide operations to add and remove methods, from which override could be defined [11]. However, we feel that override is an important operation that

it is characterized by peculiar typing rules; we prefer to study it directly and not to explain it away at an early stage.

Similarly, the choice of objects with a fixed number of components, instead of extensible ones [1, 15, 16, 21], is a conscious one. Without ruling out future work on extensible objects, we feel that fixed-size objects are easier to handle, particularly in the later stages of our type-theoretical development.

Finally, we do not provide an operation to extract a method from an object as a function. As we shall see, such an operation is incompatible with object subsumption in typed calculi. Methods are inseparable from objects and cannot be recovered as functions; this consideration inspired the use of a specialized ς-notation instead of the familiar λ-notation for parameters.

1.2 Derived Semantics

An inspection of the primitive semantics reveals close similarities between objects and records of functions. It is natural then to try to define objects in terms of records and functions. The correct definition is, however, not evident; we describe a few options.

All implementations of standard (single-dispatch) object-oriented languages are based on self-application so we examine this option first. In the self-application semantics [14], methods are functions, objects are records, object selection is record selection plus self-application, and update is simply update. Records themselves can be encoded as functions over a domain of labels. We write $\langle l_i=a_i\ ^{i\in 1..n}\rangle$ for the record with labels l_i and fields a_i; we write $r\cdot l_j$ for record selection (extracting the l_j component of r), and $r\cdot l_j:=b$ for functional update (producing a copy of r with the l_j component replaced by b).

Self-application semantics

For $o \equiv [l_i=\varsigma(x_i)b_i\ ^{i\in 1..n}]$ (l_i distinct)

o	≜	$\langle l_i=\lambda(x_i)b_i\ ^{i\in 1..n}\rangle$			
$o.l_j$	≜	$o\cdot l_j(o)$	=	$b_j\{x_j\leftarrow o\}$	$(j\in 1..n)$
$o.l_j \Leftarrow \varsigma(y)b$	≜	$o\cdot l_j:=\lambda(y)b$	=	$[l_j=\varsigma(y)b,\ l_i=\varsigma(x_i)b_i\ ^{i\in(1..n)-\{j\}}]$	$(j\in 1..n)$

The simple equalities shown above, based on the standard β-reduction of λ-terms, reveal that the self-application semantics matches the primitive semantics. Hence, untyped objects can be faithfully interpreted using λ-abstraction, application, and record constructions. Records can be further reduced to pure λ-terms. If we are concerned only with untyped object calculi, little else needs to be said.

Unfortunately, the match between primitive and self-application semantics does not directly extend to typed calculi. By interpreting ς-binders directly as λ-binders, the self-application semantics causes the type of each method to be contravariant in the host object type. The contravariance then blocks expected subtyping relations. For example, the object type Point with x,y integer fields is interpreted as the recursive record type Point = ⟨x,y: Point→Int⟩, where we write $\langle l_i:B_i\ ^{i\in 1..n}\rangle$ for the type of records with labels l_i and types B_i ($i\in 1..n$). But the type Point = ⟨x,y: Point→Int⟩ does not include as a subtype ColorPoint = ⟨x,y: ColorPoint→Int, c: ColorPoint→Color⟩, which is the interpretation of the type of points with color.

In view of these difficulties with typing, alternative encodings of objects have been investigated. One immediate idea is to try to "hide" the self parameters so that their types do not appear in contravariant position. This can be achieved by the use of recursive definitions, binding all the self parameters recursively to the object itself [9]. Note that ς is not interpreted as λ in this semantics:

Recursive-record semantics

For $o \equiv [l_i=\varsigma(x_i)b_i\ ^{i\in 1..n}]$ (l_i distinct)				
o	$\triangleq$	$\mu(x)\langle l_i=b_i\{x_i\leftarrow x\}\ ^{i\in 1..n}\rangle$	$=$	$\langle l_i=b_i\{x_i\leftarrow o\}\ ^{i\in 1..n}\rangle$
$o.l_j$	$\triangleq$	$o\cdot l_j$	$=$	$b_j\{x_j\leftarrow o\}$ $\quad (j\in 1..n)$

Here, the semantics of method invocation matches the primitive semantics. Moreover, the expected subtyping relations are validated, because the troublesome contravariant parameters are "hidden". If we require only an object calculus without override, then recursive-record semantics will do. Unfortunately, override does not work as expected in this semantics; the most plausible definition appears to be the following:

$$o.l_j \Leftarrow \varsigma(y)b \triangleq \mu(x)\, o\cdot l_j:=(\lambda(y)b)(x) \quad (j\in 1..n)$$
$$= \langle l_j=b\{y\leftarrow(o.l_j\Leftarrow\varsigma(y)b)\},\ l_i=b_i\{x_i\leftarrow o\}\ ^{i\in(1..n)-\{j\}}\rangle$$
$$\not\equiv \mu(x)\langle l_j=b\{y\leftarrow x\},\ l_j=b_j\{x_j\leftarrow x\}\ ^{i\in(1..n)-\{j\}}\rangle$$

The result of overriding a method is no longer in the form of an object: record update fails to update "inside the μ" so that all the other methods can become aware of the update. Given the above semantics we have, for example:

$o \triangleq [l_1=\varsigma(x)3,\ l_2=\varsigma(x)x.l_1]$
$p \triangleq o.l_1\Leftarrow\varsigma(x)5$

with primitive semantics	$p.l_1 = 5$	$p.l_2 = 5$
with recursive-record semantics	$p.l_1 = 5$	$p.l_2 = 3$

Therefore, the recursive-record semantics is not adequate. We can argue that the fixpoint operator has been used too soon, and that override has no chance of working once recursion is frozen. In contrast, recursion in the self-application semantics is open-ended: self-application occurs only at method invocation time, not at object-construction time.

As an attempt to salvage the recursive-record semantics, and its nice typing properties, we can try to delay the application of the μ in that semantics with a λ. Objects are then "record generators" that can be converted to recursive records, when needed, by applying a fixpoint operator (**Y**). In this way, we obtain the generator semantics of objects [12]:

Generator semantics

For $o \equiv [l_i=\varsigma(x_i)b_i\ ^{i\in 1..n}]$ (l_i distinct)				
o	$\triangleq$	$\lambda(x)\langle l_i=b_i\{x_i\leftarrow x\}\ ^{i\in 1..n}\rangle$		
$o.l_j$	$\triangleq$	$\mathbf{Y}(o)\cdot l_j$	$=$ $b_j\{x_j\leftarrow\mathbf{Y}(o)\}$	$\not\equiv$ $b_j\{x_j\leftarrow o\}$
$(j\in 1..n)$				
$o.l_j\Leftarrow\varsigma(y)b$	$\triangleq$	$\lambda(x)\, o\cdot l_j:=(\lambda(y)b)(x)$	$=$ $[l_j=\varsigma(y)b,\ l_i=\varsigma(x_i)b_i\ ^{i\in(1..n)-\{j\}}]$	
$(j\in 1..n)$				

This semantics works for update, but does not match the primitive semantics for method invocation. Put in the best light, it distinguishes between records and record generators, to the effect that method execution can be performed only after the application of **Y**, and method override only before. For example, after a method invocation returning self, it is no longer possible to perform an update. As a special case, an object cannot update itself through its own methods. Since self-update on fields is a common and important operation, one has to introduce a distinction between field update and method update in order to allow at least the former.

Split-self semantics is a combination of all of the above techniques. It separates the fields from the proper methods of an object. Collectively, the fields represent the state of the object. The

methods take the state as a parameter (as in the self-application semantics), and are bound to each other by recursion (as in the recursive-record semantics). Proper method override is not permitted. Objects and object generators are distinguished (as in the generator semantics); here we describe only the object part.

Split-self semantics

For $o \equiv [l_i=\varsigma(x)b_i\ ^{i\in 1..n}, l_j=\varsigma(x)b_j\ ^{j\in n+1..p}]$ (l_i,l_j distinct; the b_i are fields, that is, do not use x)					
o	$\triangleq$	$\langle s=\langle l_i=b_i\ ^{i\in 1..n}\rangle, m=\mu(xm)\langle l_i=\lambda(xs)b_j'\ ^{i\in 1..n}\rangle$		(for appropriate b_j')	
$o.l_i$	$\triangleq$	$o\cdot s\cdot l_i$	=	b_i	($i\in 1..n$)
$o.l_j$ ($j\in n+1..p$)	$\triangleq$	$o\cdot m\cdot l_j(o\cdot s)$	=	$b_j'\{xs\leftarrow o\cdot s, xm\leftarrow o\cdot m\}$	
$o.l_i:=b$	$\triangleq$	$\langle s=(o\cdot s\cdot l_i:=b), m=o\cdot m\rangle$			($i\in 1..n$)

The separation of state from methods causes self to split into two parts. Each method body b_j must be transformed into an appropriate new body b_j', as follows: for state extraction through self, $x.l_i$ becomes $xs.l_i$; for method invocation through self, $x.l_j$ becomes $xm.l_j$; for other uses of self, x must be repackaged into $\langle s=xs, m=xm\rangle$. These transformations are hard to perform mechanically, in general, but Hofmann and Pierce have studied type-driven techniques [13]. If these transformations are performed correctly, split-self semantics implements the primitive semantics, except for method override.

One advantage of this rather complex encoding is that it corresponds well to the behavior of class-based languages. Moreover, it can be extended to a typed encoding [19]. Finally, split-self semantics becomes considerably simpler with imperative update, since the splitting and repackaging of self is no longer necessary.

In summary, it seems hard to find simple, general, and correct encodings by syntactic means. Semantically, however, we can resort to the richer vocabulary of type constructions available in models. Specifically, the denotational semantics we give in [2] is a self-application semantics where, for example, we interpret the type Point as the union of all the solutions to the equations of the form $X = \langle x,y:X\rightarrow Int, \ldots\rangle$, including for example $X = \langle x,y:X\rightarrow Int\rangle$ and $X = \langle x,y:X\rightarrow Int, c:X\rightarrow Color\rangle$. With this definition, ColorPoint is forced to be a subtype of Point. The denotational semantics provides a justification for our type theories and our equational theories, and in part guided us in their choice.

In the core of this paper we take objects as a primitive notion, and we give them direct typing rules and equational theories. No further attempt is made to encode typed objects as typed λ-terms. However, we still strive to find the simplest object calculus from which more complex object calculi may be syntactically derived.

1.3 Paper Outline

In section 2 we study the untyped calculus sketched in section 1.1. We discover that we are able to encode the λ-calculus in it. We show three interesting untyped examples. In section 3 we study first-order systems. In section 4 we introduce subsumption, in the form of object subtyping. Using recursion we provide typings for the untyped examples. We conclude by noting that recursively defined object types do not allow some desirable subsumptions, and we outline some solutions. The appendix lists the typing and equational rules.

2. Untyped Calculi

In this section we investigate untyped object calculi. We assume the primitive semantics as the intended semantics of objects, and formalize it. We also show how to express functions and fixpoint operators in terms of objects. Finally, we discuss some untyped examples.

2.1 The Untyped ς-calculus

The following formal syntax describes a pure object calculus without functions. In later sections, this calculus is the basis for our typed calculi.

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

Here, an object $[l_i=\varsigma(x_i)b_i\ ^{i\in 1..n}]$ has method names l_i and methods $\varsigma(x_i)b_i$. In a method ς(x)b, x is the self variable and b is the body.

Notation

- $o.l_j:=b$ stands for $o.l_j \Leftarrow \varsigma(y)b$, for an unused y. We call $o.l_j:=b$ an *update* operation.
- [..., l=b, ...] stands for [..., l=ς(y)b, ...], for an unused y. We call l=b a *field*.
- We identify ς(x)b with ς(y)(b{x←y}), for any y not occurring free in b.

To complete the formal syntax of the ς-calculus we give the definitions of free variables (FV) and substitution (b{x←a}) for ς-terms.

Object scoping and substitution

FV(ς(y)b)	≜	FV(b)-{y}
FV(x)	≜	{x}
$FV([l_i=\varsigma(x_i)b_i\ ^{i\in 1..n}])$	≜	$\bigcup^{i\in 1..n} FV(\varsigma(x_i)b_i)$
FV(o.l)	≜	FV(o)
FV(o.l⇐ς(y)b)	≜	FV(o) ∪ FV(ς(y)b)

(ς(y)b){x←a}	≜	ς(y')(b{y←y'}{x←a})	for y' ∉ FV(ς(y)b)∪FV(a)∪{x}
x{x←a}	≜	a	
y{x←a}	≜	y	for y ≠ x
$[l_i=\varsigma(x_i)b_i\ ^{i\in 1..n}]\{x\leftarrow a\}$	≜	$[l_i=(\varsigma(x_i)b_i)\{x\leftarrow a\}\ ^{i\in 1..n}]$	
(o.l){x←a}	≜	(o{x←a}).l	
(o.l⇐ς(y)b){x←a}	≜	(o{x←a}).l⇐((ς(y)b){x←a})	

In Definition 2.1-1, below, we capture the primitive semantics of objects. This definition sets out three reduction relations: top-level one-step reduction (↛), one-step reduction (→), and general many-step reduction (↠). As is customary, we do not make error conditions explicit. We simply

assume that objects and methods are used consistently, and that otherwise computations stop without producing a result (so, for example, [].l does not rewrite to anything).

Definition 2.1-1 (Reduction relations)

(1) We write $a \to b$ if for some $o \equiv [l_i = \varsigma(x_i)b_i^{\ i \in 1..n}]$ and $j \in 1..n$, either:
$a \equiv o.l_j$ and $b \equiv b_j\{x \leftarrow o\}$, or
$a \equiv o.l_j \Leftarrow \varsigma(x)c$ and $b \equiv [l_j = \varsigma(x)c, l_i = \varsigma(x_i)b_i^{\ i \in 1..n-\{j\}}]$.

(2) We write $a \rightarrow b$ if $a \equiv C[a']$, $b \equiv C[b']$, and $a' \to b'$, where C is any context.

(3) We write $\twoheadrightarrow$ for the reflexive and transitive closure of $\rightarrow$.

□

We can derive an untyped equational theory from the untyped reduction rules. The equality relation is the reflexive, transitive, and symmetric closure of $\rightarrow$. It is formalized by a set of rules:

Equational theory

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

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

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

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

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

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

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

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

A Church-Rosser theorem connects equality ($\leftrightarrow$) with reduction ($\twoheadrightarrow$):

Theorem 2.1-1 (Church-Rosser)

The relation $\twoheadrightarrow$ is Church-Rosser, and if $\vdash a \leftrightarrow b$ then there exists c such that $a \twoheadrightarrow c$ and $b \twoheadrightarrow c$.

□

The proof of this result follows the method of Tait and Martin-Löf [7]. The sequence of definitions and lemmas is standard.

Finally, we define a deterministic reduction system for the closed terms of the ς-calculus. Our intent is to describe an evaluation strategy of the sort commonly used in programming languages. A characteristic of such evaluation strategies is that they do not work under binders. In our setting, this means that when given an object $[l_i = \varsigma(x_i)b_i^{\ i \in 1..n}]$ we defer simplifying the body b_i until l_i is invoked.

The purpose of the reduction system is to reduce every expression to a result. A result is itself an expression, not subject to further reduction. For the pure ς-calculus, we define a *result* to be a

term of the form $[l_i=\varsigma(x_i)b_i\ ^{i\in 1..n}]$. (If we had constants such as natural numbers we would naturally include them among the results.)

We are interested in an evaluation strategy that does not operate under ς-binders, analogous to the weak reduction strategy of the λ-calculus. In the λ-calculus, weak reduction proceeds by reducing the function part of an application until it becomes an abstraction; then the argument is substituted into the abstraction either without evaluation, for call-by-name, or after evaluation, for call-by-value. The distinction between call-by-name and call-by-value is not so crisp for our object calculus.

This weak reduction relation is denoted $\rightsquigarrow$. It is axiomatized with three rules.

(Red Object) (where $v \equiv [l_i=\varsigma(x_i)b_i\ ^{i\in 1..n}]$)

$$\frac{}{\vdash v \rightsquigarrow v}$$

(Red Select) (where $v' \equiv [l_i=\varsigma(x_i)b_i\ ^{i\in 1..n}]$)

$$\frac{\vdash a \rightsquigarrow v' \qquad \vdash b_j\{x_j\leftarrow v'\} \rightsquigarrow v \qquad j\in 1..n}{\vdash a.l_j \rightsquigarrow v}$$

(Red Override)

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

If $\vdash a \rightsquigarrow v$, then v is a result. We observe that the reduction system is deterministic, so if $\vdash a \rightsquigarrow v$ and $\vdash a \rightsquigarrow v'$ then $v \equiv v'$. We say that a reduces to v, or that v is the result of a.

The first rule says that results are not reduced further. The second rule says that in order to evaluate an expression $a.l_j$, we should first calculate the result of a, check whether it is in the form $[l_i=\varsigma(x_i)b_i\ ^{i\in 1..n}]$ with $j \in 1..n$, and then evaluate $b_j\{x_j\leftarrow[l_i=\varsigma(x_i)b_i\ ^{i\in 1..n}]\}$. The third rule says that in order to evaluate an expression $a.l_j \Leftarrow \varsigma(x)b$, we should first calculate the result of a, check whether it is in the form $[l_i=\varsigma(x_i)b_i\ ^{i\in 1..n}]$ with $j \in 1..n$, and return $[l_j=\varsigma(x)b,\ l_i=\varsigma(x_i)b_i\ ^{i\in(1..n)-\{j\}}]$. (Note that we do not compute inside b or inside the b_i.)

The next proposition says that $\rightsquigarrow$ is sound with respect to $\twoheadrightarrow$:

Proposition 2.1-1 (Soundness of weak reduction)

If $\vdash a \rightsquigarrow v$ then $a \twoheadrightarrow v$, and hence $\vdash a \leftrightarrow v$.

□

This proposition can be checked with a trivial induction on the structure of the proof that $\vdash a \rightsquigarrow v$.

Further, we would like $\rightsquigarrow$ to be complete with respect to $\twoheadrightarrow$, in the following sense:

Conjecture 2.1-1 (Completeness of weak reduction)

Let a be a closed term and v be a result. If $a \twoheadrightarrow v$ then there exists v' such that $\vdash a \rightsquigarrow v'$.

□

We do not attempt to prove this conjecture here. Standard methods developed for the λ-calculus should be applicable, however.

The rules immediately suggest an algorithm for reduction. The algorithm takes a term and, if it converges, produces a result or the token *wrong*, which represents a dynamic type error. We write Outcome(c) for the outcome of running the algorithm on input c, assuming the algorithm terminates. The algorithm can be defined as follows:

$$\text{Outcome}([l_i=\varsigma(x_i)b_i\ ^{i\in 1..n}]) \triangleq [l_i=\varsigma(x_i)b_i\ ^{i\in 1..n}]$$

Outcome($a.l_j$) ≜
  let o = Outcome(a)
  in if o is of the form $[l_i=\varsigma(x_i)b_i\ ^{i\in 1..n}]$ with $j \in 1..n$
      then Outcome($b_j\{x_j\leftarrow o\}$)
      else *wrong*

Outcome($a.l_j \Leftarrow \varsigma(x)b$) ≜
  let o = Outcome(a)
  in if o is of the form $[l_i=\varsigma(x_i)b_i\ ^{i\in 1..n}]$ with $j \in 1..n$
      then $[l_j=\varsigma(x)b,\ l_i=\varsigma(x_i)b_i\ ^{i\in(1..n)-\{j\}}]$
      else *wrong*

Clearly, $\vdash c \rightsquigarrow v$ if and only if Outcome(c) = v and v is not *wrong*.

In section 3.2.2 and 4.1.2, we consider $\rightsquigarrow$ again and study its properties related to typing.

2.2 Functions as Objects

It would be possible to add ordinary λ-terms to our object calculus. However, having two similar variables binders (λ and ς) in a small calculus seems excessive. We could replace ς with λ, identifying methods with functions. We feel, instead, that the ς-binders have a special status. First, as we have seen, they can be given a simple and direct rewrite semantics, instead of an indirect semantics involving both λ-abstraction and application. Second, ς-binders by themselves have a surprising expressive power; we show below that they can be as expressive as the λ-binders.

We define a translation ⟪ ⟫ from λ-terms to pure objects:

Translation of the untyped λ-calculus

⟪x⟫ ≜ x

⟪b(a)⟫ ≜ ⟪b⟫ • ⟪a⟫ where p • q ≜ (p.arg := q).val

⟪λ(x)b{x}⟫ ≜
  [arg = ς(x) x.arg,
   val = ς(x) ⟪b{x}⟫{x←x.arg}]

The idea here is that an application ⟪b(a)⟫ first stores the argument ⟪a⟫ into a known place (the field arg, whose initial value is unimportant) inside of ⟪b⟫, and then invokes a method of ⟪b⟫ that can access the argument through self. For example:

⟪(λ(x)x)(y)⟫ ≡ ([arg = ς(x) x.arg, val = ς(x) x.arg].arg:=y).val = y

Note that the translation maps nested λ's to nested ς's: although every method has a single self parameter, we can emulate functions with multiple parameters.

Renaming (α-conversion) of λ-bound variables is valid under the translation (up to α-conversion) because of the renaming properties of ς-binders. We can verify that β-conversion is valid under the translation as well:

let o ≡ [arg = ⟪a⟫, val = ς(x) ⟪b{x}⟫{x←x.arg}]

⟪(λ(x)b{x})(a)⟫
  ≡ ([arg = ς(x) x.arg, val = ς(x) ⟪b{x}⟫{x←x.arg}].arg:=⟪a⟫).val
  = o.val = (⟪b{x}⟫{x←x.arg}){x←o} = ⟪b{x}⟫{x←o.arg}
  = ⟪b{x}⟫{x←⟪a⟫} = ⟪b{a}⟫

However, η-conversion is not valid under the translation, since not every object has the form [arg = ..., val = ...].

This translation can be extended to provide a natural interpretation of λ-terms with default parameters and with call-by-keyword. We write λ(x=c)b{x} for a function with a single parameter x with default c. We write f(a) for a normal application of f to a, and f() for an application of f to its default. For example, (λ(x=c)x)() = c and (λ(x=c)x)(a) = a. The interpretation of λ-terms with a single default parameter is:

$$\langle\!\langle \lambda(x=c)b\{x\} \rangle\!\rangle \triangleq [arg = \langle\!\langle c \rangle\!\rangle, val = \varsigma(x)\, \langle\!\langle b\{x\} \rangle\!\rangle \{x \leftarrow x.arg\}]$$

$$\langle\!\langle b(a) \rangle\!\rangle \triangleq \langle\!\langle b \rangle\!\rangle \bullet \langle\!\langle a \rangle\!\rangle$$

$$\langle\!\langle b() \rangle\!\rangle \triangleq \langle\!\langle b \rangle\!\rangle .val$$

We write $\lambda(x_i=c_i\ ^{i\in 1..n})b$ for a function with multiple parameters x_i with defaults c_i. The application $f(y_i=a_i\ ^{i\in 1..m})$ can provide fewer parameters than f expects, and in any order. The association of the provided actuals to the formals is made by the names x_i, with the defaults being used for the missing actuals. The interpretation of λ-terms with call-by-keyword and default parameters is:

$$\langle\!\langle \lambda(x_i=c_i\ ^{i\in 1..n})b\{x_i\ ^{i\in 1..n}\} \rangle\!\rangle \triangleq \qquad x_i \neq val,\ z \notin FV(b)$$
$$[x_i = \langle\!\langle c_i \rangle\!\rangle^{\,i\in 1..n},\ val = \varsigma(z)\, \langle\!\langle b\{x_i\ ^{i\in 1..n}\} \rangle\!\rangle \{x_i \leftarrow z.x_i\ ^{i\in 1..n}\}]$$

$$\langle\!\langle b(y_i=a_i\ ^{i\in 1..m}) \rangle\!\rangle \triangleq (\langle\!\langle b \rangle\!\rangle .y_i := \langle\!\langle a_i \rangle\!\rangle^{\,i\in 1..m}).val \qquad y_i \neq val$$

2.3 Fixpoints

As a consequence of the translation of λ-terms into pure objects, we obtain object-oriented versions of all the encodings that are possible within the λ-calculus. None of these encodings seem, however, particularly inspiring; more direct object-oriented encodings can usually be found. Such is the case, for example, for fixpoint operators. The encoding given below is much simpler than others that can be obtained by translation of λ-terms:

A fixpoint operator

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

We can verify the fixpoint property as follows, recalling that p•q ≡ (p.arg:=q).val is the encoding of function application:

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

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

In particular, if we add a constant *fix* to the λ-calculus, and set $\langle\!\langle fix \rangle\!\rangle \triangleq$ fix, then $\langle\!\langle fix(f) \rangle\!\rangle = \langle\!\langle f(fix(f)) \rangle\!\rangle$.

Furthermore, we can provide a translation for μ(x)b{x} that is more compact than its natural definition as $\langle\!\langle fix(\lambda(x)b\{x\}) \rangle\!\rangle$:

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

with the unfolding property:

$$
\begin{aligned}
&\langle\!\langle \mu(x)b\{x\} \rangle\!\rangle \\
&\quad \equiv [\mathrm{rec} = \varsigma(x)\, \langle\!\langle b\{x\} \rangle\!\rangle \{x \leftarrow x.\mathrm{rec}\}].\mathrm{rec} \\
&\quad = \langle\!\langle b\{x\} \rangle\!\rangle \{x \leftarrow x.\mathrm{rec}\} \{x \leftarrow [\mathrm{rec} = \varsigma(x)\, \langle\!\langle b\{x\} \rangle\!\rangle \{x \leftarrow x.\mathrm{rec}\}]\} \\
&\quad \equiv \langle\!\langle b\{x\} \rangle\!\rangle \{x \leftarrow [\mathrm{rec} = \varsigma(x)\, \langle\!\langle b\{x\} \rangle\!\rangle \{x \leftarrow x.\mathrm{rec}\}].\mathrm{rec}\} \\
&\quad \equiv \langle\!\langle b\{x\} \rangle\!\rangle \{x \leftarrow \langle\!\langle \mu(x)b\{x\} \rangle\!\rangle\} \\
&\quad \equiv \langle\!\langle b\{\mu(x)b\{x\}\} \rangle\!\rangle .
\end{aligned}
$$

The recursion operators described above use only object primitives. With λ-abstraction and application, we can write the fixpoint operator of Mitchell *et al.* [16]:

$$\mathrm{fix}' \triangleq \lambda(f)\, [\mathrm{rec} = \varsigma(s)\, f(s.\mathrm{rec})].\mathrm{rec}$$

whose translation is similar, but not identical, to our fix:

$$\langle\!\langle \mathrm{fix}' \rangle\!\rangle \equiv [\mathrm{arg} = \varsigma(x)x.\mathrm{arg},\ \mathrm{val} = \varsigma(x)\, [\mathrm{rec} = \varsigma(s)\, ((x.\mathrm{arg}).\mathrm{arg} := s.\mathrm{rec}).\mathrm{val}].\mathrm{rec}\ ]$$

2.4 Examples

The general problem we are confronting is to find useful type systems for the untyped ς-calculus. We now examine examples that can be easily written in the untyped ς-calculus, but pose interesting typing difficulties. Our examples involve updating self, so neither the recursive-record semantics nor the generator semantics would be adequate. One of the examples is based on overriding a proper method, so the split-self semantics does not apply. The typing requirements range from very basic ones, like typing a calculator object, to very sophisticated ones, like typing an object-oriented version of the numerals inspired by Scott numerals [20].

In these examples we freely use numeric constants, and we use λ-terms since we have seen that they can be encoded.

2.4.1 Backup Methods

We now give a simple example that illustrates two techniques: storing self and using two different versions of self at once. We define an object that is able to keep backup copies of itself, for example as an auditing trail. This object has a backup method, and a retrieve method that returns the last backup:

Objects with backup

$$
\begin{aligned}
o \triangleq [&\mathrm{retrieve} = \varsigma(s_1)\, s_1\ , \\
&\mathrm{backup} = \varsigma(s_2)\, s_2.\mathrm{retrieve} \Leftarrow \varsigma(s_1)\, s_2\ , \\
&\textit{(additional fields and methods)}\ ]
\end{aligned}
$$

The initial retrieve method is set to return the initial object. Whenever the backup method is invoked, it stores a copy of self into retrieve. Note that backup stores the self s_2 that is current at backup-invocation time, not the self s_1 that will be current at retrieve-invocation time. For example:

$$o' \triangleq o.\mathrm{backup} = [\mathrm{retrieve} = \varsigma(s_1)\, o, \ldots]$$

Later, possibly after modifying the additional fields and methods, we can extract the object that was most recently backed up. The retrieve method returns the self that was current at the last invocation of backup, as desired.

$$o'.\mathrm{retrieve} = o$$

We can cascade invocations of the retrieve method to recover older and older backups, eventually converging to the initial object.

2.4.2 Object-Oriented Natural Numbers

The technique of storing self, illustrated in the previous example, comes up in another interesting situation. We would like to define the object-oriented natural numbers, that is, objects that respond to the methods iszero (test for zero), pred (predecessor), and succ (successor), and behave like the natural numbers.

We need to define only the numeral zero, since its succ method will generate all the other numerals. Obviously, the numeral zero should answer true to the iszero question, and zero to the pred question.

The succ method is not so easy to manufacture. When succ is invoked on zero, it should modify zero so that it becomes one. That is, succ should modify self so that it answers false to the iszero question, and zero to the pred question. Moreover, succ should be such that when invoked again on numeral one, it produces an appropriate numeral two, etc. Hence, for any numeral, succ should update iszero to answer false, and should update pred to return the self that is current when succ is invoked.

```
zero ≜
    [iszero = true,
     pred = ς(x) x
     succ = ς(x) (x.iszero := false).pred := x ]
```

Here the body of succ consists of two cascaded updates to self. We can verify, with a few tests, that the operational semantics of natural numbers is well represented.

Instead of defining numbers with iszero, pred, and succ, we can define numbers with only two methods: succ and case. This gives a simpler, although more opaque, encoding of the natural numbers that does not depend on booleans:

Object-oriented numerals

```
zero ≜
    [case = λ(z) λ(s) z,
     succ = ς(x) x.case := λ(z) λ(s) s(x) ]
```

The case method is in fact a field containing a function of two arguments. For a numeral n, the first argument is returned if n is zero, otherwise the second argument is applied to the predecessor of n; that is, n.case(a)(f) equals a if n is zero, and equals f(x) if n is nonzero and x is its predecessor. The predecessor of n is obtained by the now familiar technique of capturing a previous self. We can compute:

```
one  ≜  zero.succ  =  [case = λ(z) λ(s) s(zero),  succ = (unchanged)]
two  ≜  one.succ   =  [case = λ(z) λ(s) s(one),   succ = (unchanged)]
```

Moreover, we can recover the iszero and pred methods as functions:

```
iszero  ≜  λ(n) n.case(true)(λ(p)false)
pred    ≜  λ(n) n.case(zero)(λ(p)p)
```

In this example, as in the one in the previous section, current self and future self can be statically nested and handled at once. Because of this, it is critical that self be a named parameter. Providing a single keyword “self” in the scope of a method, as done in many object-oriented languages, would not be sufficient.

2.4.3 A Calculator

Our third example is that of a calculator object. We exploit the ability to override methods to record the pending arithmetic operation. When an operation add or sub is entered, the equals method is overridden with code for addition or subtraction. The first two components (arg, acc) are needed for the internal operation of the calculator, while the other four (enter, add, sub, equals) provide the user interface. A reset operation could be added to clear the state and restore the initial equals method.

Calculator

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

This definition is slightly subtle; it is meant to provide the following behavior:

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

3. First-Order Calculi

We now begin to investigate the type theory of the ς-calculus. We start with a simple first-order type system with object types.

We compose our typed systems from *formal system fragments* (collected in the appendix). Each fragment is named Δ_s for an appropriate subscript *s*. These fragments can be reassembled to form standard typed calculi. Each fragment consists of a set of related rules. Each rule has a number of antecedent judgments above a horizontal line and a single conclusion judgment below the line. Each judgment has the form $E \vdash \mathcal{J}$, for a typing environment E and an assertion $\mathcal{J}$ depending on the judgment. An antecedent of the form "$E,E_i \vdash \mathcal{J}_i \quad \forall i \in 1..n$" is intended as an abbreviation for n antecedents "$E,E_1 \vdash \mathcal{J}_1$ ... $E,E_n \vdash \mathcal{J}_n$" if n>0, and if n=0 for "$E \vdash \diamond$", which means that E is well-formed. Instead, a rule containing "$j \in 1..n$" indicates that there are n separate rules, one for each j. Each rule has a name whose first word is determined by the kind of judgment in its conclusion; for example names of the form "(Type ...)" are for rules whose conclusion is a type judgment.

3.1 The Object Fragment

We start with the formal system fragment corresponding to object types. An object of type $[l_i:B_i \ ^{i \in 1..n}]$ can be formed from a collection of n methods whose self parameters have type $[l_i:B_i \ ^{i \in 1..n}]$ and whose bodies have types B_1 ... B_n. We always assume, when writing $[l_i:B_i \ ^{i \in 1..n}]$, that the l_i are distinct and that permutations do not matter. When a method l_i is invoked, it produces a result having the corresponding type B_i. A method can be overridden while preserving the type of its host object.

Two judgments are used below: type judgments $E \vdash B$ (meaning that B is a well-formed type in the environment E) and value judgments $E \vdash b : B$ (meaning that b has type B in E). Environments contain typing assumptions for variables.

Δ_{Ob}

(Type Object)

$$\frac{E \vdash B_i \quad \forall i \in 1..n \qquad (l_i \text{ distinct})}{E \vdash [l_i : B_i \ ^{i \in 1..n}]}$$

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

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

(Val Select)

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

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

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

Notation

- $o.l_j := b$ stands for $o.l_j \Leftarrow \varsigma(y : A) b$, for an appropriate A and $y \notin FV(b)$.
- $[..., l = b, ...]$ stands for $[..., l = \varsigma(y : A) b, ...]$, for an appropriate A and $y \notin FV(b)$.
- $[..., l, m : B, ...]$ stands for $[..., l : B, m : B ...]$, in examples.
- We identify $\varsigma(x : A) b$ with $\varsigma(y : A)(b\{x \leftarrow y\})$, for any $y \notin FV(b)$.

The ς-bound variables have type annotations equal to the type of their host object. An object type $[l_i : B_i \ ^{i \in 1..n}]$ exhibits only the result types B_i of its methods: it does not explicitly list the types of the ς-bound variables. The types of all these variables are equal to the object type itself, so no information is missing. The definitions of free variables and substitution are similar to the ones in section 2.1.

3.2 A Complete Calculus

We can obtain a well-rounded typed version of the ς-calculus by adding to Δ_{Ob} two standard fragments for variables and for a constant type. Furthermore, we can include a fragment for typed λ-terms. In section 4.6 we deal with recursion.

The environment judgment $E \vdash \diamond$ is used to construct well-formed environments for variables:

Δ_x

(Env ø)

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

(Env x)

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

(Val x)

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

The fragment Δ_K introduces a ground type K.

Δ_K

(Type Const)

$$\frac{E \vdash \diamond}{E \vdash K}$$

Function types are described in the following fragment:

$\Delta_{\to}$

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

We now define the calculi:

$\mathbf{Ob_1}$	$\triangleq$	$\Delta_K \cup \Delta_x \cup \Delta_{Ob}$	the first-order typed ς-calculus
$\mathbf{F_1}$	$\triangleq$	$\Delta_K \cup \Delta_x \cup \Delta_{\to}$	the first-order typed λ-calculus
$\mathbf{FOb_1}$	$\triangleq$	$\Delta_K \cup \Delta_x \cup \Delta_{\to} \cup \Delta_{Ob}$	the first-order typed λς-calculus

It may seem puzzling that no elements are given for the ground type K. The standard use for K is as a starting point for building function types, such as a type of Church numerals $(K \to K) \to (K \to K)$ [7]. A ground type is not strictly necessary as a starting point for building object types, because the type [] is available. However, we include Δ_K in $\mathbf{Ob_1}$ to simplify comparisons with $\mathbf{F_1}$.

For the rest of section 3.2, we concentrate on $\mathbf{Ob_1}$, studying its basic properties.

3.2.1 Unique Types

The $\mathbf{Ob_1}$ calculus enjoys an important property: every term has a unique type.

Proposition 3.2.1-1 ($\mathbf{Ob_1}$ has unique types)
If $E \vdash a : A$ and $E \vdash a : A'$ are derivable in $\mathbf{Ob_1}$, then $A \equiv A'$.
□

The proof is a trivial induction on the derivation of $E \vdash a : A$, and extends to $\mathbf{FOb_1}$.

Unique typing is an obvious property for $\mathbf{Ob_1}$, but small perturbations of the rules do not always preserve it. The property remains true if we omit the type annotation for override, by adopting the rule:

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

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

However, it fails if we omit type annotations for object construction, by adopting the rule:

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

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

For example, in the modified system, $[l = \varsigma(x)x.l]$ has type $[l:A]$ for any A. Still, the convention of omitting ς-binders entirely for methods that do not depend on self is innocuous. For example, [l=3] has unique type [l:Int] if Int is the type of 3.

3.2.2 Subject Reduction

The weak reduction relation $\rightsquigarrow$ of section 2.1 can be extended to $\mathbf{Ob_1}$ terms. For this purpose, we simply ignore and carry along any type information:

(Red Object) (where $v \equiv [l_i = \varsigma(x_i:A_i)b_i \ ^{i \in 1..n}]$)

$$\frac{}{\vdash v \rightsquigarrow v}$$

(Red Select) (where $v' \equiv [l_i=\varsigma(x_i:A_i)b_i\ ^{i\in 1..n}]$)

$$\frac{\vdash a \rightsquigarrow v' \qquad \vdash b_j\{x_j \leftarrow v'\} \rightsquigarrow v \qquad j\in 1..n}{\vdash a.l_j \rightsquigarrow v}$$

(Red Override)

$$\frac{\vdash a \rightsquigarrow [l_i=\varsigma(x_i:A_i)b_i\ ^{i\in 1..n}] \qquad j \in 1..n}{\vdash a.l_j \Leftarrow \varsigma(x:A)b \rightsquigarrow [l_j=\varsigma(x:A_j)b,\ l_i=\varsigma(x_i:A_i)b_i\ ^{i\in(1..n)-\{j\}}]}$$

In **Ob_1**, reduction preserves types. Technically, we have the following subject reduction result.

Theorem 3.2.2-1 (Subject reduction for Ob_1)

Let c be a closed term and v be a result, and assume $\vdash c \rightsquigarrow v$. If $\emptyset \vdash c : C$ then $\emptyset \vdash v : C$.

Proof

The proof is by induction on the derivation of $\vdash c \rightsquigarrow v$.

(Red Object) This case is trivial, since c = v.

(Red Select) Suppose $\vdash a.l_j \rightsquigarrow v$ because $\vdash a \rightsquigarrow [l_i=\varsigma(x_i:A_i)b_i\ ^{i\in 1..n}]$ and $\vdash b_j\{x_j\leftarrow[l_i=\varsigma(x_i:A_i)b_i\ ^{i\in 1..n}]\} \rightsquigarrow v$. Assume that $\emptyset \vdash a.l_j : C$. Then $\emptyset \vdash a : A$ for some A of the form $[l_j:C, ...]$. By induction hypothesis, we have $\emptyset \vdash [l_i=\varsigma(x_i:A_i)b_i\ ^{i\in 1..n}] : A$. This implies that all A_i equal A and that $\emptyset$, $x_j : A \vdash b_j : C$. By a standard substitution lemma, it follows that $\emptyset \vdash b_j\{x_j\leftarrow[l_i=\varsigma(x_i:A)b_i\ ^{i\in 1..n}]\} : C$. By induction hypothesis, we obtain $\emptyset \vdash v : C$.

(Red Override) Suppose $\vdash a.l_j \Leftarrow \varsigma(x:A)b \rightsquigarrow [l_j=\varsigma(x:A_j)b,\ l_i=\varsigma(x_i:A_i)b_i\ ^{i\in(1..n)-\{j\}}]$ because $\vdash a \rightsquigarrow [l_i=\varsigma(x_i:A_i)b_i\ ^{i\in 1..n}]$. Assume that $\emptyset \vdash a.l_j \Leftarrow \varsigma(x:A)b : C$. Then C equals A and $\emptyset \vdash a : A$. In addition, since $\emptyset \vdash a.l_j \Leftarrow \varsigma(x:A)b : A$, we obtain also $\emptyset$, $x:A \vdash b : B$, with A of the form $[l_j:B,...]$. By induction hypothesis, we have $\emptyset \vdash [l_i=\varsigma(x_i:A_i)b_i\ ^{i\in 1..n}] : A$. This implies that A must have the form $[l_j:B, l_i:B_i\ ^{i\in(1..n)-\{j\}}]$. It follows that A_i equals A and $\emptyset$, $x_i:A \vdash b_i:B_i$ for all i. Thus, $\emptyset \vdash [l_j=\varsigma(x:A)b, l_i=\varsigma(x_i:A)b_i\ ^{i\in(1..n)-\{j\}}] : A$, that is, $\emptyset \vdash [l_j=\varsigma(x:A)b, l_i=\varsigma(x_i:A)b_i\ ^{i\in(1..n)-\{j\}}] : C$.

□

The algorithm for reduction of section 2.1 is extended for type annotations:

Outcome($[l_i=\varsigma(x_i:A_i)b_i\ ^{i\in 1..n}]$) ≜
  $[l_i=\varsigma(x_i:A_i)b_i\ ^{i\in 1..n}]$

Outcome($a.l_j$) ≜
  let o = Outcome(a)
  in if o is of the form $[l_i=\varsigma(x_i:A_i)b_i\ ^{i\in 1..n}]$ with $j \in 1..n$
    then Outcome($b_j\{x_j\leftarrow o\}$)
    else *wrong*

Outcome($a.l_j \Leftarrow \varsigma(x:A)b$) ≜
  let o = Outcome(a)
  in if o is of the form $[l_i=\varsigma(x_i:A_i)b_i\ ^{i\in 1..n}]$ with $j \in 1..n$
    then $[l_j=\varsigma(x:A_j)b,\ l_i=\varsigma(x_i:A_i)b_i\ ^{i\in(1..n)-\{j\}}]$
    else *wrong*

We obtain:

Theorem 3.2.2-2 (Ob_1 reductions cannot go wrong)

If $\emptyset \vdash c : C$ and Outcome(c) is defined, then Outcome(c) ≠ *wrong*.

□

The proof is by induction on the execution of Outcome(c), and is very similar to the proof of Theorem 3.2.2-1.

These results show the soundness of $\mathbf{Ob_1}$ typing with respect to reduction. Their proof is a good sanity check, and an introduction to similar arguments for more complex calculi. Subject reduction properties can probably be obtained for all our calculi, but we will consider them only for pure object calculi.

3.3 Equational Theory of Ob_1

We now investigate the equational theory of the Δ_{Ob} fragment, which was implicitly assumed in some of the previous discussion. The equational theories for the other fragments are standard. For simplicity, when assembling a calculus we list only the typing fragments. The corresponding equational fragments are assumed from context since, at least in this paper, they are uniquely determined.

We use a new judgment $E \vdash b \leftrightarrow c : A$ to assert that b and c are equivalent when considered as elements of type A. The first two rules for this new judgment express symmetry and transitivity; reflexivity will be obtained as a derived rule.

$\Delta_{=}$

(Eq Symm)

$$\frac{E \vdash a \leftrightarrow b : A}{E \vdash b \leftrightarrow a : A}$$

(Eq Trans)

$$\frac{E \vdash a \leftrightarrow b : A \qquad E \vdash b \leftrightarrow c : A}{E \vdash a \leftrightarrow c : A}$$

In examples we assume equational rules for constants, but we do not present these rules formally. There is an obvious rule for variables: a limited form of reflexivity.

$\Delta_{=x}$

(Eq x)

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

The first three rules for objects are congruence rules, establishing that two expressions are equal when all their corresponding subexpressions are equal. The next two rules are evaluation rules for selection and override, corresponding to the operational semantics of the untyped calculus of section 2.1.

$\Delta_{=Ob}$

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

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

(Eq Select)

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

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

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

(Eval Select) (where $A \equiv [l_i:B_i\ ^{i\in 1..n}]$, $a \equiv [l_i = \varsigma(x_i:A)b_i\ ^{i\in 1..n}]$)

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

(Eval Override) (where $A \equiv [l_i:B_i\ ^{i\in 1..n}]$, $a \equiv [l_i = \varsigma(x_i:A)b_i\ ^{i\in 1..n}]$)

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

This small equational theory is already quite interesting. In record calculi we can safely assume that two records are equal if they have the same labels and if all their corresponding components are equal. This property does not hold for objects: two objects may yield equal result for all their methods, and still be distinguishable. Consider the following objects:

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

We might think that a and b are equal at type A because $\emptyset \vdash a.x \leftrightarrow b.x : Nat$ and $\emptyset \vdash a.f \leftrightarrow b.f : Nat$. But to prove their equality, the (Eq Object) rule requires showing $\emptyset, s:A \vdash 1 \leftrightarrow s.x : Nat$. This cannot be obtained because we have no assumptions about the value of self, in particular that s.x is currently 1. In fact, self may change and invalidate this assumption about its value. For example, it is possible to distinguish a from b after updating them both with equal values:

$$\begin{aligned} a':A &\triangleq a.x:=2 \\ b':A &\triangleq b.x:=2 \end{aligned}$$

Now we have $\emptyset \vdash a'.f \leftrightarrow 1 : Nat$ while $\emptyset \vdash b'.f \leftrightarrow 2 : Nat$, so asserting $\emptyset \vdash a \leftrightarrow b : A$ would lead to a contradiction.

This example illustrates a fundamental difference between the equational theories of object calculi and record calculi, as well as a fundamental difficulty in reasoning about objects. Still, we would like to say more about object equivalence than $\Delta_{=Ob}$ allows. For example, we may wish to determine that a and b above are interchangeable in contexts that read or modify only their x components; that is, in contexts where a and b are considered as having type [x:Nat]. This equation involves an implicitly or explicit assumption that a longer object belongs to a shorter type. We examine this idea in section 4.

3.4 Functions as Objects

The $\mathbf{Ob_1}$ calculus is sufficient to encode $\mathbf{F_1}$, along the lines sketched in section 2.2. Hence, $\mathbf{FOb_1}$ has a built-in redundancy, although a very convenient one. The translation from $\mathbf{F_1}$ to $\mathbf{Ob_1}$ is shown below. Strictly speaking, this translation is defined on type derivations, but for simplicity we write it as a translation of type-annotated λ-terms.

Translation of the first-order λ-calculus

$\rho \in Var \to \mathbf{Ob_1}\text{-term}$

$$\begin{aligned} \emptyset(x) &\triangleq x \\ (\rho\{y \leftarrow a\})(x) &\triangleq \text{if } x=y \text{ then a else } \rho(x) \end{aligned}$$

$$\begin{aligned} &\langle\!\langle \emptyset \rangle\!\rangle \triangleq \emptyset \\ &\langle\!\langle E,x:A \rangle\!\rangle \triangleq \langle\!\langle E \rangle\!\rangle, x:\langle\!\langle A \rangle\!\rangle \end{aligned}$$

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

It is not difficult to verify that the translation maps valid derivations in $\mathbf{F_1}$ to valid derivations in $\mathbf{Ob_1}$. Typed β-reduction is satisfied by this encoding, but typed η-reduction is not.

In what follows, we describe several extensions of $\mathbf{F_1}$ and $\mathbf{Ob_1}$. Consider a pair of extensions, $\mathbf{F}_*$ and $\mathbf{Ob}_*$. We say that $\mathbf{Ob}_*$ can encode $\mathbf{F}_*$ when there exists a translation mapping derivations of $\mathbf{F}_*$ that do not use the first-order η rule into derivations of $\mathbf{Ob}_*$. In this sense, $\mathbf{Ob_1}$ can encode $\mathbf{F_1}$.

In addition, $\mathbf{Ob_1}$ can encode an extension of $\mathbf{F_1}$ with a fixpoint operator $\mathrm{fix}_A{:}(A \to A) \to A$ for each A, in such a way that $\langle\!\langle \mathrm{fix}_A(f) \rangle\!\rangle = \langle\!\langle f(\mathrm{fix}_A(f)) \rangle\!\rangle$. All we need is the fixpoint operator of section 2.3, which is typable within $\mathbf{Ob_1}$.

$$
\begin{array}{l}
\langle\!\langle \mathrm{fix}_A \rangle\!\rangle \rho \triangleq \\
\quad [\mathrm{arg} = \varsigma(x{:}\langle\!\langle (A \to A) \to A \rangle\!\rangle)\, x.\mathrm{arg}, \\
\quad \mathrm{val} = \varsigma(x{:}\langle\!\langle (A \to A) \to A \rangle\!\rangle)\, ((x.\mathrm{arg}).\mathrm{arg} := x.\mathrm{val}).\mathrm{val}]
\end{array}
$$

Both $\langle\!\langle \mathrm{fix}_A \rangle\!\rangle \rho$ and the ς-bound variables of this term have type

$$\langle\!\langle (A \to A) \to A \rangle\!\rangle = [\mathrm{arg}{:}\ [\mathrm{arg}{:}\ \langle\!\langle A \rangle\!\rangle, \mathrm{val}{:}\ \langle\!\langle A \rangle\!\rangle], \mathrm{val}{:}\ \langle\!\langle A \rangle\!\rangle].$$

$\mathbf{Ob_1}$, unlike $\mathbf{F_1}$, is not normalizing; this is obvious from the definability of fixpoint operators. In fact, there exist simple divergent terms. For example $[l{=}\varsigma(x{:}[l{:}[]])x.l].l$ is typable as follows:

$\emptyset \vdash \diamond$	by (Env ∅)
$\emptyset \vdash []$	by (Type Object) with n=0
$\emptyset \vdash [l{:}[]]$	by (Type Object) with n=1
$\emptyset, x{:}[l{:}[]] \vdash \diamond$	by (Env x)
$\emptyset, x{:}[l{:}[]] \vdash x : [l{:}[]]$	by (Val x)
$\emptyset, x{:}[l{:}[]] \vdash x.l : []$	by (Val Select)
$\emptyset \vdash [l{=}\varsigma(x{:}[l{:}[]])x.l] : [l{:}[]]$	by (Val Object) with n=1
$\emptyset \vdash [l{=}\varsigma(x[l{:}[]])x.l].l : []$	by (Val Select)

4. First-Order Calculi with Subsumption

A characteristic of object-oriented languages is that an object can emulate another object that has fewer methods, since the former supports the entire protocol of the latter. Conversely, a context that expects an object with a given method protocol can be filled with an object that has a more extended protocol. We call this notion *subsumption*: an object can subsume another object that has a more limited protocol.

No object calculus can fully justify its existence without some notion of subsumption. This criticism should first be directed to $\mathbf{FOb_1}$, studied in section 3. We should notice that, in accordance with the self-application semantics, there is no difficulty in adding to $\mathbf{FOb_1}$ a new operation that extracts a method from an object and returns it as a function with parameter self.

Without a good reason for ruling out this extraction operation, an object calculus would be just an oddly restricted record calculus. As it happens, the idea of extracting a method from an object is uncharacteristic of object-oriented languages and, as we shall see shortly, this is precisely because this operation is fundamentally incompatible with the typing of subsumption.

To address this inadequacy, in this section we define a particular form of subsumption that is induced by a subtyping relation between object types. An object that belongs to a given object type also belongs to any supertype of that type, and can subsume objects in the supertype.

4.1 A Calculus with Subtyping

We begin with the basic rules of subtyping: reflexivity, transitivity, and subsumption. It is also convenient to add a type constant, Top, that is a supertype of every type. The judgment $E \vdash A <: B$ asserts that A is a subtype of B in environment E.

$\Delta_{<:}$

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

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

(Val Subsumption)
$$\frac{E \vdash a : A \quad E \vdash A <: B}{E \vdash a : B}$$

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

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

After these general preliminaries, we write the subtyping rules for function and object types:

$\Delta_{<:\rightarrow}$

(Sub Arrow)
$$\frac{E \vdash A' <: A \quad E \vdash B <: B'}{E \vdash A \rightarrow B <: A' \rightarrow B'}$$

$\Delta_{<:\mathbf{Ob}}$

(Sub Object)
$$\frac{E \vdash B_i \quad \forall i \in 1..n+m}{E \vdash [l_i : B_{i\ i \in 1..n+m}] <: [l_i : B_{i\ i \in 1..n}]} \qquad (l_i \text{ distinct})$$

The subtyping rule for function types is the standard one. A function type $A \rightarrow B$ is *contravariant* in its domain, so $A \rightarrow B <: A' \rightarrow B$ requires $A' <: A$, and is *covariant* in its codomain, so $A \rightarrow B <: A \rightarrow B'$ requires $B <: B'$.

The subtyping rule for objects allows a longer object type $[l_i : B_{i\ i \in 1..n+m}]$ to be a subtype of a shorter object type $[l_i : B_{i\ i \in 1..n}]$. Moreover, an object type is *invariant* in its component types: $[l_i : B_{i\ i \in 1..n+m}] <: [l_i : B_i'_{\ i \in 1..n}]$ requires $B_i \equiv B_i'$ for all $i \in 1..n$. That is, object types are neither covariant nor contravariant; in particular, $[l : A \rightarrow B]$ is neither covariant in B nor contravariant in A; this is necessary for soundness (see section 4.5.1).

We define the calculi:

$$\mathbf{Ob}_{1<:} \triangleq \mathbf{Ob}_1 \cup \Delta_{<:} \cup \Delta_{<:\mathbf{Ob}}$$
$$\mathbf{F}_{1<:} \triangleq \mathbf{F}_1 \cup \Delta_{<:} \cup \Delta_{<:\rightarrow}$$
$$\mathbf{FOb}_{1<:} \triangleq \mathbf{FOb}_1 \cup \Delta_{<:} \cup \Delta_{<:\mathbf{Ob}} \cup \Delta_{<:\rightarrow}$$

Note that the translation of $\mathbf{F_1}$ into $\mathbf{Ob_1}$ does not extend to a corresponding translation of $\mathbf{F_{1<:}}$ into $\mathbf{Ob_{1<:}}$, because $\langle\!\langle A \to B \rangle\!\rangle = [\text{arg}:\langle\!\langle A \rangle\!\rangle, \text{val}:\langle\!\langle B \rangle\!\rangle]$ is invariant in $\langle\!\langle A \rangle\!\rangle$ and $\langle\!\langle B \rangle\!\rangle$. Hence, $\mathbf{Ob_{1<:}}$ is essentially a restricted version of $\mathbf{FOb_{1<:}}$ with invariant function types. We can recover the covariant/contravariant subtyping properties of function types in a pure object calculus by an encoding based on bounded universal and existential types [3].

4.1.1 Minimum Types

With the addition of subsumption we have obviously lost the unique-types property of $\mathbf{Ob_1}$ (see section 3.2.1). However, a weaker property holds: every term of $\mathbf{Ob_{1<:}}$ has a minimum type. This property also holds for $\mathbf{F_{1<:}}$, and we believe that it holds for $\mathbf{FOb_{1<:}}$.

In order to prove the minimum-types property for $\mathbf{Ob_{1<:}}$, we consider a system $\mathbf{MinOb_{1<:}}$ obtained from $\mathbf{Ob_{1<:}}$ by removing (Val Subsumption), and by modifying the (Val Object) and (Val Override) rules as follows:

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

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

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

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

Typing in $\mathbf{MinOb_{1<:}}$ is unique, as we show below. We can easily extract from $\mathbf{MinOb_{1<:}}$ a typechecking algorithm that, given an environment E and a term a, computes a type A such that $E \vdash a : A$ if one exists.

The next three propositions are proved by easy inductions on the derivations of $E \vdash a : A$.

Proposition 4.1.1-1 ($\mathbf{MinOb_{1<:}}$ typings are $\mathbf{Ob_{1<:}}$ typings)

If $E \vdash a : A$ is derivable in $\mathbf{MinOb_{1<:}}$, then it is also derivable in $\mathbf{Ob_{1<:}}$.

□

Proposition 4.1.1-2 ($\mathbf{MinOb_{1<:}}$ has unique types)

If $E \vdash a : A$ and $E \vdash a : A'$ are derivable in $\mathbf{MinOb_{1<:}}$, then $A \equiv A'$.

□

Proposition 4.1.1-3 ($\mathbf{MinOb_{1<:}}$ has smaller types than $\mathbf{Ob_{1<:}}$)

If $E \vdash a : A$ is derivable in $\mathbf{Ob_{1<:}}$ then $E \vdash a : A'$ is derivable in $\mathbf{MinOb_{1<:}}$ for some A' such that $E \vdash A' <: A$ is derivable (in either system).

□

Proposition 4.1.1-4 ($\mathbf{Ob_{1<:}}$ has minimum types)

In $\mathbf{Ob_{1<:}}$, 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 $E \vdash B <: A'$.

Proof

Assume $E \vdash a : A$. By Proposition 4.1.1-3, $E \vdash a : B$ is derivable in $\mathbf{MinOb_{1<:}}$ for some B such that $E \vdash B <: A$. By Proposition 4.1.1-1, $E \vdash a : B$ is also derivable in $\mathbf{Ob_{1<:}}$. If $E \vdash a : A'$, then $E \vdash a : B'$ is also derivable in $\mathbf{MinOb_{1<:}}$ for some B' such that $E \vdash B' <: A'$. By Proposition 4.1.1-2, $B \equiv B'$, so $E \vdash B <: A'$.

□

Just like lack of annotations for ς-binders destroys the unique-types property for $\mathbf{Ob_1}$, it destroys the minimum-types property for $\mathbf{Ob_{1<:}}$. For example, let:

$A = [l: []]$

$A' = [l: A]$

$a = [l = \varsigma(x)[l = \varsigma(x)[]]]$

then:

$\emptyset \vdash a:A \quad \text{and} \quad \emptyset \vdash a:A'$

but A and A' have no common subtype. This example also shows that minimum typing is lost for objects with fields (where the ς-binders are omitted entirely), since a can be written as [l = [l = []]] with our conventions.

The term a.l:=[] typechecks using a:A but not using a:A'. Naive type inference algorithms might find the type A' for a, and fail to find any type for a.l:=[]. Thus, the absence of minimum typings poses practical problems for type inference. Palsberg has described an ingenious type inference algorithm that surmounts these problems [18].

In contrast, in $\mathbf{Ob_{1<:}}$ (with annotations), $[l=\varsigma(x:A)[l=\varsigma(x:A)[]]] : A$ and $[l=\varsigma(x:A')[l=\varsigma(x:A)[]]] : A'$ are minimum typings.

4.1.2 Subject Reduction

As in $\mathbf{Ob_1}$ (see section 3.2.2), typing and reduction are consistent in $\mathbf{Ob_{1<:}}$. We have a subject reduction theorem:

Theorem 4.1.2-1 (Subject reduction for $\mathbf{Ob_{1<:}}$)

Let c be a closed term and v be a result, and assume $\vdash c \searrow v$. If $\emptyset \vdash c : C$ then $\emptyset \vdash v : C$.

Proof

The proof is by induction on the derivation of $\vdash c \searrow v$.

(Red Object) This case is trivial, since c = v.

(Red Select) Suppose $\vdash a.l_j \searrow v$ because $\vdash a \searrow [l_i=\varsigma(x_i:A_i)b_i \ ^{i\in 1..n}]$ and $\vdash b_j\{x_j \leftarrow [l_i=\varsigma(x_i:A_i)b_i \ ^{i\in 1..n}]\} \searrow v$. Assume that $\emptyset \vdash a.l_j : C$. Then $\emptyset \vdash a : A$ for some A of the form $[l_j:B_j, ...]$ with $\emptyset \vdash B_j <: C$. By induction hypothesis, we have $\emptyset \vdash [l_i=\varsigma(x_i:A_i)b_i \ ^{i\in 1..n}] : A$. This implies that there exists A' such that $\emptyset \vdash A' <: A$, that all A_i equal A', that $\emptyset \vdash [l_i=\varsigma(x_i:A')b_i \ ^{i\in 1..n}] : A'$, and that $\emptyset, x_j : A' \vdash b_j : B_j$. By a standard substitution lemma, it follows that $\emptyset \vdash b_j\{x_j \leftarrow [l_i=\varsigma(x_i:A')b_i \ ^{i\in 1..n}]\} : B_j$. By induction hypothesis, we obtain $\emptyset \vdash v : B_j$ and, by subsumption, $\emptyset \vdash v : C$.

(Red Override) Suppose $\vdash a.l_j \Leftarrow \varsigma(x:A)b \searrow [l_j=\varsigma(x:A_j)b, l_i=\varsigma(x_i:A_i)b_i \ ^{i\in(1..n)-\{j\}}]$ because $\vdash a \searrow [l_i=\varsigma(x_i:A_i)b_i \ ^{i\in 1..n}]$. Assume that $\emptyset \vdash a.l_j \Leftarrow \varsigma(x:A)b: C$. Then $\emptyset \vdash a.l_j \Leftarrow \varsigma(x:A)b: A$ and $\emptyset \vdash A <: C$. In addition, since $\emptyset \vdash a.l_j \Leftarrow \varsigma(x:A)b : A$, we obtain also $\emptyset, x : A \vdash b : B$, with A of the form $[l_j:B,...]$. By induction hypothesis, we have $\emptyset \vdash [l_i=\varsigma(x_i:A_i)b_i \ ^{i\in 1..n}] : A$. This implies that A_j has the form $[l_j:B, l_i:B_i \ ^{i\in(1..n)-\{j\}}]$, that $\emptyset \vdash A_j <: A$, that A_i equals A_j, and that $\emptyset, x_i: A_j \vdash b_i : B_i$ for all i. By a standard bound weakening lemma, it follows that $\emptyset, x:A_j \vdash b : B$. Therefore, $\emptyset \vdash [l_j=\varsigma(x:A_j)b, l_i=\varsigma(x_i:A_j)b_i \ ^{i\in(1..n)-\{j\}}] : A_j$. We obtain $\emptyset \vdash [l_j=\varsigma(x:A_j)b, l_i=\varsigma(x_i:A_j)b_i \ ^{i\in(1..n)-\{j\}}] : C$ by subsumption.

□

As in $\mathbf{Ob_1}$, the proof of subject reduction is simply a sanity check. It remains an easy proof, with just one subtle point: notice that the proof would have failed if we had defined (Red Override) so that $\vdash a.l_j \Leftarrow \varsigma(x:A)b \searrow [l_j=\varsigma(x:A)b, l_i=\varsigma(x_i:A_i)b_i \ ^{i\in(1..n)-\{j\}}]$ with an A instead of an A_j in the bound for x.

For the reduction algorithm, we still obtain:

Theorem 4.1.2-2 ($Ob_{1<:}$ reductions cannot go wrong)

If $\varnothing \vdash c : C$ and Outcome(c) is defined, then Outcome(c) $\neq$ *wrong*.

□

4.2 Programming in $Ob_{1<:}$

Both $Ob_{1<:}$ and $FOb_{1<:}$ are fairly useful object calculi with subsumption, although they are limited by the absence of recursive type definitions. In this section we show a simple example of their use.

We begin by defining typed versions of one-dimensional and two-dimensional points:

P_x	$\triangleq$	[x:Real]	one-dimensional points on the x axis
P_y	$\triangleq$	[y:Real]	one-dimensional points on the y axis
P_{xy}	$\triangleq$	[x,y:Real]	two-dimensional points

We obtain a multiple-subtyping situation: $P_{xy} <: P_x$ and $P_{xy} <: P_y$.

We can enrich this example with proper methods, by extending two-dimensional points with polar coordinates:

P_{xyrt}	$\triangleq$	[x,y,r,t: Real] two-dimensional points with redundant coordinates
p	$\triangleq$	[x=0, y=1, r=ς(s:P_{xyrt})sqrt(s.x^2+s.y^2), t=ς(s:P_{xyrt})atan2(s.y,s.x)]
to-polar	$\triangleq$	λ(o:P_{xyrt})[x=ς(s:P_{xyrt})cos(s.t)*s.r, y=ς(s:P_{xyrt})sin(s.t)*s.r, r=o.r, t=o.t]
to-cart	$\triangleq$	λ(o:P_{xyrt})[x=o.x, y=o.y, r=ς(s:P_{xyrt})sqrt(s.x^2+s.y^2), t=ς(s:P_{xyrt})atan2(s.y,s.x)]

In p the cartesian coordinates are primitive, and the polar coordinates are computed by methods. The function to-polar maps a point to itself, except that the polar coordinates become primitive and the cartesian coordinates become derived; to-cart works in the opposite direction. Thus, to-polar and to-cart convert passive data to active computation, and vice versa.

We may want to calculate to-polar(p) before performing computations on p that are more efficient in polar representation. However, from the point of view of computing a correct result, there is no harm in not knowing which representation is primitive for a point, as long as we maintain the invariant that the cartesian and the polar coordinates represent the same point.

Since $P_{xyrt} <: P_{xy}$ we can pass a P_{xyrt} element q to a client that operates only with the cartesian coordinates. If we pass to-cart(q) instead of q, we can be certain that client updates to x and y will not break the representation invariant.

4.3 Classes and Inheritance

The object-oriented notions of *class* and *inheritance* are not explicit in our calculi. Here, we discuss how these notions can be represented. Our discussion is rather informal and relies on common object-oriented jargon. To avoid ambiguity, we call pre-methods those functions that become methods once embedded into objects.

We take the point of view that "inheritance" means pre-method reuse, and that "classes" are collections of interdependent reusable pre-methods. As in Modula-3 [17], we make methods reusable by writing them first as functions (that is, as pre-methods), and then by repeatedly embedding these functions into objects.

The key idea is that if $A \equiv [l_i:B_i\ ^{i\in 1..n}]$ is an object type, then:

Class(A)	$\triangleq$	[new:A, l_i:A→$B_i\ ^{i\in 1..n}$]
$class_A$	$\triangleq$	[new=ς(c:Class(A)) [l_i=ς(s:A)c.l_i(s) $^{i\in 1..n}$], l_i=λ(s:A)b_i{s} $^{i\in 1..n}$]

can be seen as a class type and a class. A class is an object that groups pre-methods together with a "new" operation that generates instances. The pre-methods in a class are fields that can be extracted and reused to form other classes. A type $[l_i:A\to B_i\ ^{i\in 1..k}]$, for k<n, can be seen as an abstract-class type. An element of such a type must be completed with new and the missing pre-methods before it can be instantiated.

We would like to say that a class Class(A) with more pre-methods inherits (or may inherit) from a class Class(A') with fewer pre-methods, possibly reusing some pre-methods of Class(A'). However, it can be seen easily that A<:A' does not imply Class(A)<:Class(A'). Therefore, we define an ad-hoc inheritance relation ≤ on class types that captures the intuition of method reuse. We set:

$$\text{Class}(A) \le \text{Class}(A') \quad \text{iff} \quad A <: A' \qquad (\le \text{ means "may inherit from"})$$

where we must have $A\equiv[l_i:B_i\ ^{i\in 1..n+m}]$ and $A'\equiv[l_i:B_i\ ^{i\in 1..n}]$. If Class(A) ≤ Class(A'), then $A'\to B_i <: A\to B_i$, because of the contravariance of function types. Hence, pre-methods of c':Class(A') may be reused in assembling c:Class(A), by subsumption.

Whenever c is defined by reassembling, extending, and modifying c', we may informally say that c is a subclass of c'. The multiple subtyping property, which holds for object types, induces multiple inheritance on class types: a class can reuse pre-methods from any of its superclasses.

Some or all of the component of a class may be hidden by subsumption. The components that are not hidden can be overridden; objects created from the resulting class will incorporate the overridden methods.

4.4 Equational Theory of $\mathbf{Ob_{1<:}}$

We extend the equational theory of $\mathbf{Ob_1}$ to take subsumption into account. First, the following equalities are associated with the $\Delta_{<:}$ fragment:

$\Delta_{=<:}$

(Eq Subsumption)
$$\frac{E \vdash a \leftrightarrow a' : A \qquad E \vdash A <: B}{E \vdash a \leftrightarrow a' : B}$$

(Eq Top)
$$\frac{E \vdash a:A \qquad E \vdash b:B}{E \vdash a \leftrightarrow b : \text{Top}}$$

Still, this does not give us enough power to compare objects of different lengths (except as members of Top). We remedy this by augmenting the $\Delta_{=Ob}$ fragment from section 3.3 with the rule (Eq Sub Object), which allows us to ignore methods that do not appear in the type of an object. At the same time we generalize (Eval Select) and (Eval Override) to deal with objects and types of different lengths:

$\Delta_{=<:Ob}$

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

(Eval Select) (where $A\equiv[l_i:B_i\ ^{i\in 1..n}]$, $a\equiv[l_i=\varsigma(x_i:A')b_i\ ^{i\in 1..n+m}]$)
$$\frac{E \vdash a : A \qquad j\in 1..n}{E \vdash a.l_j \leftrightarrow b_j\{x_j\leftarrow a\} : B_j}$$

(Eval Override) (where $A \equiv [l_i:B_i\ ^{i\in 1..n}]$, $a \equiv [l_i = \varsigma(x_i:A')b_i\ ^{i\in 1..n+m}]$)

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

According to (Eq Sub Object) an object can be truncated to its externally visible collection of methods, but only if those methods do not depend on the hidden ones.

In section 3.3 we saw that a and b below cannot be equal at type A:

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

Using (Eq Sub Object) it is at least possible to show that a and b are equal at type [x:Nat], by showing $\emptyset \vdash a \leftrightarrow [x=1]:[x:Nat]$ and $\emptyset \vdash b \leftrightarrow [x=1]:[x:Nat]$.

We can ask next whether a and b are equal at type [f:Nat]. This seems reasonable because the only way to distinguish a and b is to update their x components, which are not exposed in [f:Nat]. However, we cannot apply (Eq Object) and (Eq Sub Object) because we would need to show $\emptyset$, $s:[f:Nat] \vdash 1 \leftrightarrow s.x : Nat$. A stronger rule would be needed to show $\emptyset \vdash a \leftrightarrow b:[f:Nat]$; we leave this for future work. We note for now that, unlike the rest of the equational theory, this new equation is invalid in an imperative language, where somebody might hold a pointer to the whole object b and modify x.

4.5 Objects Versus Records

We consider two natural but incorrect extensions of $\mathbf{Ob}_{1<:}$: covariant object types and method extraction. Taken together, these two failed extensions show that, in a calculus with subsumption, object typing is fundamentally different from record typing since both extensions are sound for records. A third extension, elder, allows us to define a new version of a method in terms of the previous one.

4.5.1 Covariant Object Types

The subtyping rule for object types should be compared with the analogous rule for cartesian products. Object types are invariant in their components, while cartesian products are covariant in both of their components. Similarly, record types $\langle l_i:A_i\ ^{i\in 1..n}\rangle$, which generalize cartesian products, are covariant in all of their components.

It is natural to ask what happens if we allow object types to be covariant in their components. Suppose we adopt the following more liberal subtyping rule for objects:

(Sub Object/covariant)

$$\frac{E \vdash B_i <: B_i' \qquad E \vdash B_j \qquad \forall i \in 1..n,\ j \in n+1..m \qquad (l_i \text{ distinct})}{E \vdash [l_i:B_i\ ^{i\in 1..n+m}] <: [l_i:B_i'\ ^{i\in 1..n}]}$$

Then we can derive a contradiction. Assume PosReal<:Real and ln:PosReal→Real (the natural logarithm). Define:

P	≜	[x:Real, f:Real]	note that [x=1.0, f=ς(s:P)ln(s.x)] : P is not derivable
Q	≜	[x:PosReal, f:Real]	with Q <: P by (Sub Object/covariant)
a	≜	[x=1.0, f=ς(s:Q)ln(s.x)]	we have a:Q (Val Object), hence a:P (Val Subsumption)
b	≜	a.x:= -1.0	from a:P we have b:P (Val Override), therefore b.f:Real

Now we have $\emptyset \vdash b.f \leftrightarrow \ln(-1.0) : \text{Real}$; we can derive a typing for a program that applies a function to an argument outside its domain. Hence, the type system is unsound, as a direct consequence of adding (Sub Object/covariant). The above example can be recast with other nontrivial subtypings in place of PosReal<:Real, such as a subtyping between object types. Note that the example involves simple field update, and does not require proper method override.

The basic reason for this problem is that each method relies on the types of the other methods, through the type of self. This dependence is essentially contravariant, and hence is incompatible with covariance.

4.5.2 Method Extraction

Let us now assume we have an operation for extracting a method from an object, as discussed at the beginning of section 4:

$$\frac{E \vdash a : A \qquad j \in 1..n}{E \vdash a.l_j : A \to B_j} \quad \text{(Val Extract)} \quad (\text{where } A \equiv [l_i : B_i \ ^{i \in 1..n}])$$

$$\frac{E \vdash a : A \qquad j \in 1..n}{E \vdash a.l_j \leftrightarrow \lambda(x_j:A)b_j : A \to B_j} \quad \text{(Eval Extract)} \quad (\text{where } A \equiv [l_i : B_i \ ^{i \in 1..n}],\ a \equiv [l_i = \varsigma(x_i:A')b_i \ ^{i \in 1..n+m}])$$

Then we define:

P	≜	[x:Int, f:Int]	
p	≜	[x=1, f=1]	we have p:P (Val Object)
Q	≜	[x,y:Int, f:Int]	we have Q <: P (Sub Object)
a	≜	[x=1, y=1, f=ς(s:Q)s.x+s.y]	we have a:Q (Val Object), hence also a:P (Val Subsumption)
b	≜	a.f	we have b:P→Int (Val Extract), with $\emptyset \vdash b \leftrightarrow \lambda(s:P)s.x+s.y : P \to \text{Int}$ (Eval Extract)

Now we have $\emptyset \vdash b(p) \leftrightarrow (\lambda(s:P)s.x+s.y)(p) \leftrightarrow p.x+p.y : \text{Int}$, via (Eval Beta), but p does not have a y component. If we change an A to A' in the conclusion of (Eval Extract), which becomes $E \vdash a.l_j \leftrightarrow \lambda(x_j:A')b_j : A \to B_j$, then the (Eval Beta) step is not even possible.

Hence, it is unsound to add the method extraction operation to $\mathbf{FOb_{1<:}}$ (or to $\mathbf{Ob_{1<:}}$, with encoded function types), although it is sound to add it to $\mathbf{FOb_1}$ (or to $\mathbf{Ob_1}$).

4.5.3 Elder

Although in general it is unsound to extract a method, it is sound to refer to the previous value of a method in the course of an override. Just like the new value of a record field can be defined from its old value, the new code for a method can reuse the overridden code. We write $a.l_j \Leftarrow \varsigma(x:A,y:B_j)b$ for the result of overriding the l_j method of a with $\varsigma(x:A,y:B_j)b$; when l_j is invoked, x is self and y is the body of the old method. We call y *elder*.

We give only the type and evaluation rules for override with elder:

$$\frac{E \vdash a : A \qquad E, x:A, y:B_j \vdash b : B_j \qquad j \in 1..n}{E \vdash a.l_j \Leftarrow \varsigma(x:A,y:B_j)b : A} \quad \text{(Val Override with elder)} \quad (\text{where } A \equiv [l_i : B_i \ ^{i \in 1..n}])$$

(Eval Override with elder) (where $A \equiv [l_i:B_i\ ^{i\in 1..n}]$, $a \equiv [l_i=\varsigma(x_i:A')b_i\ ^{i\in 1..n+m}]$, $x_j \notin dom(E)$)

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

Override with elder could easily be implemented from method extraction. But, unlike method extraction, override with elder is sound because we never apply the old method to an arbitrary element of type A. The old method's self remains bound to the object's self, of type A'.

In a calculus like ours, with primitive objects but without primitive classes, this mechanism provides a form of inheritance by reusing methods of existing objects in new objects. Consider, for example, a two-dimensional point p with a draw method that produces a picture with the point in it. This point would have the type $P_{xyd} \triangleq [x,y:Real, draw:Bitmap]$. We may change the draw method in order to invert the color of the picture, reusing the existing drawing code: $p.draw \Leftarrow \varsigma(s:P_{xyd},e:Bitmap)invert(e)$.

For the sake of simplicity, we do not add elder to our core calculus. However, we think this should be a useful construct in practice.

4.6 Recursion

In $\mathbf{Ob_1}$ and $\mathbf{Ob_{1<:}}$ we can find typings for objects that use self, such as $[l = \varsigma(x:[l:[]])\ x.l]$ of type $[l:[]]$. However, we cannot find informative typings for objects whose methods return either self or an updated self: this feature calls for the use of recursion. Therefore, we complete our first-order system by adding rules for recursive types $\mu(X)A$ with explicit fold/unfold maps. To add recursive types to a calculus with subtyping we need to introduce type variables with subtype bounds in the environments. These bounded variables are needed in the rule for subtyping recursive types [6]. We write E,X,E' as an abbreviation for the environment E,X<:Top,E'.

$\Delta_{<:X}$

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

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

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

$\Delta_{<:\mu}$

(Type Rec<:) $\dfrac{E,X \vdash A}{E \vdash \mu(X)A}$

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

(Val Fold) $\dfrac{E \vdash a : A\{X \leftarrow \mu(X)A\}}{E \vdash fold(\mu(X)A, a) : \mu(X)A}$

(Val Unfold) $\dfrac{E \vdash a : \mu(X)A}{E \vdash unfold(a) : A\{X \leftarrow \mu(X)A\}}$

$\Delta_{=<:\mu}$

(Eq Fold)

$$\frac{E \vdash a \leftrightarrow a' : A\{X \leftarrow \mu(X)A\}}{E \vdash fold(\mu(X)A, a) \leftrightarrow fold(\mu(X)A, a') : \mu(X)A}$$

(Eq Unfold)

$$\frac{E \vdash a \leftrightarrow a' : \mu(X)A}{E \vdash unfold(a) \leftrightarrow unfold(a') : A\{X \leftarrow \mu(X)A\}}$$

(Eval Fold)

$$\frac{E \vdash a : \mu(X)A}{E \vdash fold(\mu(X)A, unfold(a)) \leftrightarrow a : \mu(X)A}$$

(Eval Unfold)

$$\frac{E \vdash a : A\{X \leftarrow \mu(X)A\}}{E \vdash unfold(fold(\mu(X)A, a)) \leftrightarrow a : A\{X \leftarrow \mu(X)A\}}$$

The (Sub Rec) rule determines the variance behavior of recursive types. If A{X,Y} is covariant in X, then the variance of Y in μ(X)A{X,Y} is the same as the variance of Y if A{X,Y}. But if A{X,Y} is contravariant in X, then μ(X)A{X,Y} is always invariant in Y (because after unfolding we obtain A{μ(X)A{X,Y},Y} with Y in positions of opposite variance). Similarly, if A{X,Y} is invariant in X then μ(X)A{X,Y} is always invariant in Y.

We obtain the calculi:

$$\mathbf{Ob}_{1<:\mu} \triangleq \mathbf{Ob}_{1<:} \cup \Delta_{<:X} \cup \Delta_{<:\mu}$$
$$\mathbf{F}_{1<:\mu} \triangleq \mathbf{F}_{1<:} \cup \Delta_{<:X} \cup \Delta_{<:\mu}$$
$$\mathbf{FOb}_{1<:\mu} \triangleq \mathbf{FOb}_{1<:} \cup \Delta_{<:X} \cup \Delta_{<:\mu}$$

$\mathbf{FOb}_{1<:\mu}$ is the strongest system we consider in this paper, and $\mathbf{Ob}_{1<:\mu}$ is a rather mild restriction of $\mathbf{FOb}_{1<:\mu}$ where only invariant function types can be encoded. $\mathbf{FOb}_{1<:\mu}$ can be shown sound by denotational methods, as discussed in the introduction. We complete the paper by providing $\mathbf{Ob}_{1<:\mu}$ typings for the untyped examples of section 2.4, and then by discussing the limitations of $\mathbf{Ob}_{1<:\mu}$ (and $\mathbf{FOb}_{1<:\mu}$).

4.7 Typing the Examples

We start with the typed version of the example from section 2.4.1. The type in question is:

$$Bk \triangleq \mu(X)[retrieve:X, backup:X, ...]$$

We obtain the following typed version of the code, where UBk ≜ [retrieve:Bk, backup:Bk, ...] is the unfolding of Bk:

```
fold(Bk,
    [ retrieve = ς(s1:UBk) fold(Bk, s1),
      backup = ς(s2:UBk) fold(Bk, s2.retrieve⇐ς(s1:UBk) fold(Bk, s2)),
      ...
    ])  :  Bk
```

If we now consider the types Point=[x,y:Int] and PointBk=μ(X)[retrieve:X, backup:X, x,y:Int], we obtain PointBk <: Point modulo an unfolding. Thus, points with backup can subsume points.

Similar techniques can be used to type the calculator example from section 2.4.3:

$$Calc = \mu(X)[arg,acc:Real, enter:Real \rightarrow X, add,sub:X, equals:Real]$$

Furthermore, we would like to obtain inclusions such as Calc <: μ(X)[enter:Real→X, add,sub:X, equals:Real], which would allow us to hide the implementation details of a calculator. Such inclusions are the subject of the next section.

The example of the numerals of section 2.4.2 requires some additions to our first-order type system. One possibility is second-order types [3]. A simpler, first-order alternative is sum types, if we are prepared to rephrase the example. We add a type construction A+B, with operations inl_{AB}:A→(A+B), inr_{AB}:B→(A+B), and if_{ABC}: (A+B)→(A→C)→(B→C)→C). Informally, A+B is the disjoint union of A and B, inl_{AB} and inr_{AB} are the obvious injections, and if_{ABC} is used to

examine elements of A+B returning elements of C. (From now on we omit the subscripts.) For convenience, we also add a type Unit with a constant unit:Unit. The numerals can be expressed using sums as follows:

Nat ≜ μ(X) [case: Unit+X, succ: X]

zero ≜
  fold(Nat,
   [case = inl(unit),
    succ = ς(x:[case:Unit+Nat, succ:Nat]) fold(Nat, x.case := inr(fold(Nat, x)))]

iszero ≜ λ(n:Nat) if(unfold(n).case)(λ(u:Unit) true)(λ(p:Nat) false)
pred ≜ λ(n:Nat) if(unfold(n).case)(λ(u:Unit) zero)(λ(p:Nat) p)

Although this code looks quite different from that of section 2.4.2, the two versions are related by a type isomorphism.

4.8 The Shortcomings of $Ob_{1<:\mu}$

The $\mathbf{Ob_{1<:\mu}}$ calculus looks very promising because it adds subtyping to a rich first-order theory. In addition to the examples in the previous section, we can use $\mathbf{Ob_{1<:\mu}}$ to write types of movable points:

P_1	≜	μ(X)[x:Int, mv_x:Int→X]	movable one-dimensional points
P_2	≜	μ(X)[x,y:Int, mv_x,mv_y:Int→X]	movable two-dimensional points

We would expect to obtain $P_2 <: P_1$. However, this is not provable, because the invariance of object types blocks the application of (Sub Rec) to the result type of mv_x.

Moreover, if we somehow allow $P_2<:P_1$, we obtain an inconsistency. Briefly, suppose we use subsumption from $p:P_2$ to $p:P_1$, and then override the mv_x method of p with one that returns a proper element of P_1. Then, some other method of p may go wrong because it assumes that mv_x produces an element of P_2. More precisely, let us define:

UP_1 ≜ [x:Int, mv_x:Int→P_1] (the unfolding of P_1)

UP_2 ≜ [x,y:Int, mv_x,mv_y:Int→P_2] (the unfolding of P_2)

$p_2 : P_2$ ≜
  fold(P_2,
    [x=ς(s_2:UP_2)unfold(s_2.mv_x(1)).y,
     y=0,
     mv_x=ς(s_2:UP_2)λ(dx:Int)fold(P_2,s_2),
     mv_y=ς(s_2:UP_2)λ(dy:Int)fold(P_2,s_2)])

$p_1 : P_1$ ≜ fold(P_1, [x=0, mv_x=ς(s_1:UP_1)λ(dx:Int)fold(P_1,s_1)])

$p : P_1$ ≜ p_2 (retyping p_2 using the assumption $P_2<:P_1$)

$q : P_1$ ≜ fold(P_1, unfold(p).mv_x := λ(dx:Int)p_1)

We have:

unfold(q).x

$\leftrightarrow$ (unfold(p_2).mv_x := λ(dx:Int)p_1).x

$\leftrightarrow$ [x=ς(s_2:UP_2)unfold(s_2.mv_x(1)).y, y=..., mv_x=λ(dx:Int)p_1, mv_y=...].x

$\leftrightarrow$ unfold(p_1).y

But unfold(p_1) does not have a y component.

As we have just seen, the failure of P_2<:P_1 is necessary. At the same time, it is unacceptable: in the common situation where a method returns an updated self, we lose all useful subsumption relations. The situation is less severe in imperative languages, where the mv_x method could be redefined to side-effect the host point and return nothing. Then, the type of the modified method would not depend on the type of self, and P_1 and P_2 would not be recursive. Even in imperative languages, though, we often find methods that, like mv_x, allocate new objects of the type of self and return them.

In many programming languages, such as Simula-67 and Modula-3, the failure of P_2<:P_1 is avoided by not allowing a subclass to change the type of a method of a superclass. For example, in our calculus, we could define mv_x to return P_1 even when embedded in P_2:

$P_1 \triangleq \mu(X)[x:Int, mv_x:Int\rightarrow X]$

$P_2' \triangleq \mu(X)[x,y:Int, mv_x:Int\rightarrow P_1, mv_y:Int\rightarrow X]$

$UP_1 \triangleq [x:Int, mv_x:Int\rightarrow P_1]$ (the unfolding of P_1)

$UP_2' \triangleq [x,y:Int, mv_x:Int\rightarrow P_1, mv_y:Int\rightarrow P_2']$ (the unfolding of P_2')

Then we have UP_2' <: UP_1, so we can at least convert every point p:P_2' to a point fold(P_1, unfold(p)) of type P_1. It is possible to strengthen the type theory to identify recursive types up to isomorphism, as in [6]; then we can obtain directly P_2'<:P_1. Whenever we invoke mv_x on an element of P_2', though, we "forget" its second dimension. For this kind of solution to be useful, Simula-67 and Modula-3, among other languages, provide dynamic testing of membership in a subtype of a given type, so that the forgotten information can be recovered. In our example, we would test for membership in the subtype P_2' of P_1.

Hence, following the approach of common object-oriented languages, we could reasonably add dynamic types [5] to $\mathbf{Ob_{1<:\mu}}$, and abandon the notion of truly static typing of subsumption. In [3] and [4] we describe alternative solutions that preserve static typing; these involve second-order constructs.

5. Related Work

We briefly review the most closely related work. There are other studies of objects that we do not discuss; in particular, several based on more or less satisfactory encodings, e.g., [10, 19].

Some ideas in our treatment of objects originated in the study of Baby Modula-3. That language resembles $\mathbf{FOb_{1<:\mu}}$ in power, although the syntactic details of the two languages are incomparable. For example, Baby Modula-3 includes a limited form of object extension, and its operational semantics has a rather strict evaluation strategy.

Bruce's TOOPL language [8] has built-in objects, and supports a form of subsumption that is obtained via two distinct subtype relations. The TOOPL semantics is based on generators, and hence distinguishes between objects and object generators (classes).

Our paper is also closely related in spirit, if not in detail, to that of Mitchell *et al.* [15, 16]. We take the same approach of defining an untyped calculus with override, based on self-application semantics, and then looking for relevant type systems. The most significant difference in outcome is that we are able to support subtyping and subsumption, along with override. In this, we have been helped by basing our calculus on fixed-size objects. On the other hand, open-ended extensible objects [11, 15] provide a more direct modeling of method inheritance.

6. Conclusions

Instead of reducing objects to more primitive concepts, we tried to capture the expected properties of objects, studying typing rules and equational theories. We developed an expressive object notation, sufficient to encode λ-calculi and to write interesting examples. We obtained an integration of subsumption and method override that has eluded formalization attempts based on encodings.

Our equational theory of objects accounts for subsumption and method override. Although the equational theory is simple and possibly incomplete, no similar theory seems to exist in the literature.

Second-order theories can be defined by extending our first-order theories with standard second-order constructs. Further work describes the second-order theories, where some deficiencies of first-order systems are remedied with an account of "Self types", and shows their soundness using a denotational semantics [2, 3].

Acknowledgments

John Mitchell was helpful during initial discussions about the subject of this paper. John Lamping suggested using sums to type the natural numbers.

Appendix A: Objects Fragments

These are the typing and equality rules for first-order objects.

Δ_{Ob}

(Type Object) (l_i distinct)

$$\frac{E \vdash B_i \quad \forall i \in 1..n}{E \vdash [l_i : B_{i\,i\in 1..n}]}$$

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

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

(Val Select)

$$\frac{E \vdash a : [l_i : B_{i\,i\in 1..n}] \quad j \in 1..n}{E \vdash a.l_j : B_j}$$

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

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

$\Delta_{<:Ob}$

(Sub Object) (l_i distinct)

$$\frac{E \vdash B_i \quad \forall i \in 1..n+m}{E \vdash [l_i : B_{i\,i\in 1..n+m}] <: [l_i : B_{i\,i\in 1..n}]}$$

$\Delta_{=Ob}$

(Eq Object) (where $A \equiv [l_i : B_{i\,i\in 1..n}]$)

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

(Eq Select)

$$\frac{E \vdash a \leftrightarrow a' : [l_i : B_{i\,i\in 1..n}] \quad j \in 1..n}{E \vdash a.l_j \leftrightarrow a'.l_j : B_j}$$

(Eq Override) (where $A \equiv [l_i : B_{i\,i\in 1..n}]$)

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

(Eval Select) (Eval Override) (in both: $A \equiv [l_i : B_{i\,i\in 1..n}]$, $a \equiv [l_i = \varsigma(x_i:A) b_{i\,i\in 1..n+m}]$)

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

$\Delta_{=<:Ob}$

(Eq Sub Object) (where $A \equiv [l_i : B_{i\,i\in 1..n}]$, $A' \equiv [l_i : B_{i\,i\in 1..n+m}]$)

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

Appendix B: Other Typing Fragments

Δ_{x}

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

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

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

Δ_{K}

(Type Const)
$$\frac{E \vdash \diamond}{E \vdash K}$$

$\Delta_{\rightarrow}$

(Type Arrow)
$$\frac{E \vdash A \quad E \vdash B}{E \vdash A \rightarrow B}$$

(Val Fun)
$$\frac{E,x:A \vdash b : B}{E \vdash \lambda(x:A)b : A \rightarrow B}$$

(Val Appl)
$$\frac{E \vdash b : A \rightarrow B \quad E \vdash a : A}{E \vdash b(a) : B}$$

$\Delta_{<:}$

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

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

(Val Subsumption)
$$\frac{E \vdash a : A \quad E \vdash A <: B}{E \vdash a : B}$$

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

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

$\Delta_{<:\rightarrow}$

(Sub Arrow)
$$\frac{E \vdash A' <: A \quad E \vdash B <: B'}{E \vdash A \rightarrow B <: A' \rightarrow B'}$$

$\Delta_{<:X}$

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

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

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

$\Delta_{<:\mu}$

(Type Rec<:)
$$\frac{E,X<:Top \vdash A}{E \vdash \mu(X)A}$$

(Sub Rec)
$$\frac{E \vdash \mu(X)A \quad E \vdash \mu(Y)B \quad E,Y<:Top,X<:Y \vdash A<:B}{E \vdash \mu(X)A <: \mu(Y)B}$$

(Val Fold)

$$\frac{E \vdash a : A\{X \leftarrow \mu(X)A\}}{E \vdash fold(\mu(X)A, a) : \mu(X)A}$$

(Val Unfold)

$$\frac{E \vdash a : \mu(X)A}{E \vdash unfold(a) : A\{X \leftarrow \mu(X)A\}}$$

Appendix C: Other Equational Fragments

$\Delta_{=}$

(Eq Symm)

$$\frac{E \vdash a \leftrightarrow b : A}{E \vdash b \leftrightarrow a : A}$$

(Eq Trans)

$$\frac{E \vdash a \leftrightarrow b : A \quad E \vdash b \leftrightarrow c : A}{E \vdash a \leftrightarrow c : A}$$

$\Delta_{=x}$

(Eq x)

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

$\Delta_{=K}$

(Eq Const)

$$\frac{k \in Op(K_{i\ i\in 1..n+1}) \quad E \vdash a_i \leftrightarrow a_i' : K_i \quad \forall i \in 1..n}{E \vdash k(a_{i\ i\in 1..n}) \leftrightarrow k(a_i'^{\ i\in 1..n}) : K_{n+1}}$$

$\Delta_{=\rightarrow}$

(Eq Fun)

$$\frac{E, x{:}A \vdash b \leftrightarrow b' : B}{E \vdash \lambda(x{:}A)b \leftrightarrow \lambda(x{:}A)b' : A \rightarrow B}$$

(Eq Appl)

$$\frac{E \vdash b \leftrightarrow b' : A \rightarrow B \quad E \vdash a \leftrightarrow a' : A}{E \vdash b(a) \leftrightarrow b'(a') : B}$$

(Eval Beta)

$$\frac{E \vdash \lambda(x{:}A)b : A \rightarrow B \quad E \vdash a : A}{E \vdash (\lambda(x{:}A)b)(a) \ \leftrightarrow\ b\{x \leftarrow a\} : B}$$

(Eval Eta)

$$\frac{E \vdash b : A \rightarrow B \quad x \notin dom(E)}{E \vdash \lambda(x{:}A)b(x) \ \leftrightarrow\ b : A \rightarrow B}$$

$\Delta_{=<:}$

(Eq Subsumption)

$$\frac{E \vdash a \leftrightarrow a' : A \quad E \vdash A <: B}{E \vdash a \leftrightarrow a' : B}$$

(Eq Top)

$$\frac{E \vdash a{:}A \quad E \vdash b{:}B}{E \vdash a \leftrightarrow b : Top}$$

$\Delta_{=<:\mu}$

(Eq Fold)

$$\frac{E \vdash a \leftrightarrow a' : A\{X \leftarrow \mu(X)A\}}{E \vdash fold(\mu(X)A, a) \leftrightarrow fold(\mu(X)A, a') : \mu(X)A}$$

(Eq Unfold)

$$\frac{E \vdash a \leftrightarrow a' : \mu(X)A}{E \vdash unfold(a) \leftrightarrow unfold(a') : A\{X \leftarrow \mu(X)A\}}$$

(Eval Fold)

$$\frac{E \vdash a : \mu(X)A}{E \vdash fold(\mu(X)A, unfold(a)) \leftrightarrow a : \mu(X)A}$$

(Eval Unfold)

$$\frac{E \vdash a : A\{X \leftarrow \mu(X)A\}}{E \vdash unfold(fold(\mu(X)A, a)) \leftrightarrow a : A\{X \leftarrow \mu(X)A\}}$$

References

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

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

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

[4] Abadi, M. and L. Cardelli, **An imperative object calculus**. *Proc. TAPSOFT'95*, 471-485. Springer-Verlag. 1995.

[5] Abadi, M., L. Cardelli, B. Pierce, and G.D. Plotkin, **Dynamic typing in a statically typed language**. *ACM Transactions on Programming Languages and Systems* **13**(2), 237-268. 1991.

[6] Amadio, R.M. and L. Cardelli, **Subtyping recursive types**. *Proc. 18th Annual ACM Symposium on Principles of Programming Languages*. 1991.

[7] Barendregt, H.P., **The lambda-calculus, its syntax and semantics**. North-Holland. 1985.

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

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

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

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

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

[13] Hofmann, M. and B.C. Pierce, **A unifying type-theoretic framework for objects**. *Proc. Symposium on Theoretical Aspects of Computer Science*. 1994.

[14] Kamin, S.N., **Inheritance in Smalltalk-80: a denotational definition**. *Proc. 15th Annual ACM Symposium on Principles of Programming Languages*, 80-87. 1988.

[15] Mitchell, J.C., **Toward a typed foundation for method specialization and inheritance**. *Proc. 17th Annual ACM Symposium on Principles of Programming Languages*, 109-124. 1990.

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

[17] Nelson, G., ed. **Systems programming with Modula-3**. Prentice Hall. 1991.

[18] Palsberg, J., **Efficient inference for object types**. *Proc. 9th Annual IEEE Symposium on Logic in Computer Science*, 186-195. 1994.

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

[20] Wadsworth, C., **Some unusual λ-calculus numeral systems.** In *To H.B. Curry: Essays on combinatory logic, lambda calculus and formalism,* J.P. Seldin and J.R. Hindley, ed. Academic Press. 1980.

[21] Wand, M., **Complete type inference for simple objects**. *Proc. 2nd Annual IEEE Symposium on Logic in Computer Science*, 37-44. 1987.