Extensible Objects Without Labels

Christopher A. Stone
Department of Computer Science
Harvey Mudd College
stone@cs.hmc.edu

Abstract

An object calculus studied in previous work with Jon Riecke [RS98] permitted both (functional) object extension and unrestricted width subtyping in a sound type system. This language allowed a simple encodings of classes as object-generators in which subclasses could be compiled and typechecked knowing only the non-private interface of the immediate superclass. However, certain encodings of mixins and more sophisticated class features were not possible.

This paper presents the Calculus of Objects and Indices (COI), a language of primitive extensible objects more analogous to tuples than to records. An object is simply a finite sequence of (unnamed) components, which are referenced by their index in the sequence. Component names are then re-introduced by taking indices as first-class values (like pointers-to-member in C++) which can be bound to variables. Since these variables — unlike record labels — can freely alphavary, all issues of name clashes disappear. Furthermore, adding new components to an object always yields an object of a strict subtype; this permits construction of nontrivial subtypes of abstract object types.

This paper demonstrates the expressive power of COI with examples that combine objects with standard type-theoretic mechanisms to encode mixins and to encode classes having the by-name subtyping of languages like C++ or Java.

1 Introduction

Early work on the foundations of object-oriented languages treated objects as particular uses of records[1]. Although it has become common to study objects as primitive concepts [AC96, FHM94], these objects typically remain syntactically record-like; each component is identified by and accessed with a unique label, the name of that field or method.

This can cause well-known problems in systems permitting new fields or methods to be added to an existing object: what happens when a new method is added to an object already containing a method with the same name? In the presence of standard width subtyping, such actions cannot be prevented at compile-time. Overriding the pre-existing method would be unsound, but a record cannot have two components with the same label.

Previous work with Jon Riecke [RS98] provided a solution with a standard type system but a non-standard representation of objects. The key innovation of this calculus (hereafter referred to as RS) was to add the idea of a "dictionary" to separate out the association between names and the corresponding object components. This ensured that extending objects never replaced existing components; at worst pre-existing methods could be "shadowed" and externally inaccessible. (For further technical details, see [RS98].) However, certain desirable encodings were still not expressible.

1.1 Issue 1: Class Types

One application for primitive objects is to model classes in conventional object-oriented languages. One difficulty is often the type hierarchies found in class-based languages. Most formal studies of objects and subtyping assume structural types; the type of an object is determined directly by the types of its components. In contrast, classes in C++ or Java act as object types, and these types have a subtyping hierarchy mirroring the inheritance graph.

Figure 1 shows an encoding of two classes, `Point` and its subclass `Point2D`, built using RS objects and a type-theoretic core in the syntax of Standard ML[2]. Each class is represented as a SML structure containing the type `T` of objects of this class and a constructor function `new` to create objects. The `Point` module also includes a "friend function" `eq` to provide functionality akin to a binary method for single-dimensional points. Thus an object created by `Point.new` has both an `x` component and a `get1` method; the former is private, being hidden in the module's signature by subsumption. (As usual, for simplicity only methods are considered here;

[1] E.g., "Object-oriented programming is based on record structures (called *objects*) intended as named collection of values (*attributes*) and functions (*methods*)." [CM89]

[2] For reference, the RS type $\{\!| l_1{:}e_1, \ldots, l_n{:}\tau_n |\!\}$ is the type of an object with components $l_1, \ldots, l_n$ having types $\tau_1, \ldots, \tau_n$ respectively. The expression `obj s.`$\{\!|\,|\!\}$ denotes an empty object with no components, the expression form $e.l$ means to invoke the method named l in the object e, and the syntax $e_1 \leftarrow\!\!+ l(s) = e_2$ means to take an object e_1 and add a new method labeled l whose code e_2 is parameterized by the variable s denoting `self` or `this`. Dictionaries are elided for simplicity because they play a critical role only in describing intermediate steps of a computation.

```
structure Point =
  struct
    type T = ⦃get1 : int⦄      (* interface type *)
    fun new x1 = (obj s.⦃⦄)    (* constructor function *)
                    ←+ x(s) = x1 : int
                    ←+ get1(s) = s.x : int
    fun eq (o1:T,o2:T) = (o1.get1 = o2.get1)
  end :>
  sig
    type T = ⦃get1 : int⦄
    val new : int -> T
    val eq  : T*T -> bool
  end

structure Point2D =
  struct
    type T = ⦃ get1 : real, get2 : real ⦄      (* interface type *)
    fun new (x1, x2) = Point.new(x1)           (* constructor function *)
                          ←+    x(s) = x2 : real
                          ←+ get2(s) = s.x : real
  end :>
  sig
    type T = ⦃ get1 : int, get2 : real ⦄
    val new : int*real -> T
  end
```

Figure 1: Class Encodings without Abstract Types

fields are modeled as constant methods.) An object of class `Point2D` is created by adding two components to an object of the superclass `Point`, resulting in four components of which two are externally accessible. Although the `Point` and `Point2D` each add an `x` component without being aware of the other, these two components nevertheless remain distinct and are accessed by their respective getter methods.

Because the definitions of the types `Point.T` and `Point2D.T` are exposed in the signature, the type system cannot distinguish an object of the `Point` class from any other object having a `get1` method returning an integer. Because of this, there is no easy way to implement friend functions or binary methods with access to private components. For example, the function `Point.eq` can be applied to any object with a `get1` method returning an integer, and hence cannot directly access the `x` field of its argument; there is no way to guarantee that such a field even exists.

The obvious way to fix this would be to make the types of the classes' objects partially abstract, specifying that the type of the objects created by the class is some (unspecified) subtype of the interface type. This is the approach taken in Modula-3 (and would be a natural extension to a SML-like module system [Cra98]). Assuming we permit bounded types in module interfaces, the `Point` structure becomes

```
structure Point =
  struct
     type T = ⦃ x : int, get1 : int ⦄
     fun new (x0:int) =  ...unchanged...
     fun eq (o1:T,o2:T) = (o1.x = o2.x)
  end :>
  sig
     type T <: ⦃ get1 : int ⦄
     val new : int -> T
     val eq  : T*T -> bool
  end
```

Now code can use `Point.T` to specify an object of this class, and the supertype `⦃ get1 : int ⦄` when only the public interface matters. The `eq` function can now access the private fields of `Point` objects because abstraction ensures that `eq` will be applied only to objects created by this class.

Unfortunately, this encoding does not permit useful inheritance [FR00]. Without knowing the exact definition of the type `Point.T`, there is no way to tell whether an object of this type still has this type after new components have been added; hence there is no way to define a corresponding implementation type `Point2D.T` guaranteed to be a subtype of `Point.T`. In fact in the RS system, the object values created by the `Point2D` class in Figure 1 have principal type

`⦃ x : real, get1 : int, get2 : real ⦄`

which is not a subtype of `⦃ x : int, get1 : int ⦄`.

Modula-3 does permit the creation of subtypes of such partially abstract object types exactly as needed here, but to my knowledge there is no formal semantics for this part of the language. The work of Abadi on Baby Modula-3 does include an operation for extending objects, but only for extending object values where all components are syntactically apparent [Aba94].

1.2 Issue 2: Typing Mixins

The RS type system also cannot give useful types to functions which extend their arguments. This makes the language unsuitable for encoding constructs such as mixins/parameterized classes.

For example, one can write a `colorize` function that adds a single method `getc` of some type `color` to any object; such a function could model a mixin. However, the type of this function would be:

$$\{\!|\ |\!\} \rightarrow \{\!|\ \texttt{getc : color}\ |\!\}.$$

That is, (by subsumption) `colorize` can be applied to any object at all but guarantees only that the resulting object has a `getc` method. Because this function extends its argument, we expect that the object returned also has copies of all the components that were in the argument object; the type system cannot express this fact.

Simply adding bounded polymorphism is no help; the best type would be

$$\forall(\alpha \preceq \{\!|\ |\!\}).\ (\alpha \rightarrow \{\!|\ \texttt{getc : color}\ |\!\})$$

which again does not express how the type of the object returned depends on the type α of the the argument object.

Cardelli and Mitchell [CM89] deal with this same problem in a language with extensible records by defining a type $\langle\!\langle \tau_1 \leftarrow l : \tau_2 \rangle\!\rangle$, the type of values which result from taking a record of type τ_1 and adding a field l of type τ_2. By adding an analogous type constructor $\tau_1 \leftarrow\!\!\!+ l{:}\tau_2$ to RS, we could give the type of `colorize` as

$$\forall(\alpha \preceq \{\!|\ |\!\}).\ \alpha \rightarrow (\alpha \leftarrow\!\!\!+ \texttt{getc:color})$$

This accurately describes the behavior of the function. Unfortunately, such types are not very flexible, because in general $\tau_1 \leftarrow\!\!\!+ l{:}\tau_2$ and τ_1 are unrelated types in the subtyping hierarchy (e.g., consider $\tau_1 = \{\!|\, \texttt{x:int}\, |\!\}$ and $\tau_1 \leftarrow\!\!\!+ \texttt{x:bool} \equiv \{\!|\, \texttt{x:bool}\, |\!\}$). Width subtyping does not extend to "forgetting" these extensions; thus, a variant definition which adds both a field `c` and a getter method `getc` could be given the very precise type

$$\forall(\alpha \preceq \{\!|\ |\!\}).\ \alpha \rightarrow (\alpha \leftarrow\!\!\!+ \texttt{c:color} \leftarrow\!\!\!+ \texttt{getc:color})$$

or the less precise type

$$\forall(\alpha \preceq \{\!|\ |\!\}).\ \alpha \rightarrow \{\!|\texttt{c:color, getc:color}|\!\}$$

or even the type

$$\forall(\alpha \preceq \{\!|\ |\!\}).\ \alpha \rightarrow \{\!|\texttt{getc:color}|\!\}$$

if we want to make the `c` component private, but we cannot get the type

$$\forall(\alpha \preceq \{\!|\ |\!\}).\ \alpha \rightarrow (\alpha \leftarrow\!\!\!+ \texttt{getc:color})$$

which both specifies the dependency and treats `c` as private. This last type specifies that an argument with a component `c` yields an output with a component `c` of the same type, whereas the code in question would *shadow* any such `c` component.

Thus, although this extension would permit more precise typings for mixins encoded as object-extending functions (or functors, if classes are structures), there are still many restrictions. A new component added like `c` here cannot safely be ignored through subsumption in the way we could hide the two private `x` fields in Figure 1.

2 The COI System

Figure 2 presents an alternative approach, the Calculus of Objects and Indices (COI). By removing labels (and RS dictionaries) from objects altogether, objects become a simple sequence of components, more like tuples than records. Object components are indexed by number, and these indices are made first-class values.

This design has two very useful consequences. First, adding a new component at the end of an object's sequence always creates an object of a subtype. Secondly, since variables — unlike labels — can freely alpha-vary, we can bind indices to variables and use the variables to refer to method components without any possibility of name clashes.

2.1 Syntax

The simplest COI type is $\{\!|\,|\!\}$, the type of the empty object (and, by subsumption, of every object). The exact type $@\tau$ classifies values whose principal (most-specific) type is *exactly* τ. The type of an object of τ_1 extended with a new component of type τ_2 is written $\tau_1 \leftarrow\!\!\!+ \tau_2$. Index values have types of the form $\tau_1 \Rightarrow \tau_2$; this type classifies indices which can be used with an object of type τ_1 to access a component of type τ_2. Finally, the type system includes type variables and bounded polymorphism; these are not required for the core object calculus but are included to represent the effect of partially-abstract types.

Two shorthands are useful for writing types. Ordinary object types (still with unlabeled components) can be represented as the type obtained by repeatedly extending the empty object type. This can be defined recursively as:

$$\{\!|\, \tau_1, \ldots, \tau_n \,|\!\} \quad := \quad (@\{\!|\, \tau_1, \ldots, \tau_{n-1} \,|\!\}) \leftarrow\!\!\!+ \tau_n$$

For example,

$$\begin{aligned}\{\!|\, \tau_1, \tau_2 \,|\!\} &= (@\{\!|\, \tau_1 \,|\!\}) \leftarrow\!\!\!+ \tau_2 \\ &= (@((@\{\!|\,|\!\}) \leftarrow\!\!\!+ \tau_1)) \leftarrow\!\!\!+ \tau_2\end{aligned}$$

It is also convenient to have a shorthand for the result of extending an object with several components:

$$\tau \leftarrow\!\!\!+ (\tau_1, \ldots, \tau_n) \quad := \quad (@(\tau \leftarrow\!\!\!+ \tau_1)) \leftarrow\!\!\!+ (\tau_2, \ldots, \tau_n)$$

Thus the type abbreviated $\{\!|\, \tau_1, \tau_2 \,|\!\}$ could also be written $(@\{\!|\,|\!\}) \leftarrow\!\!\!+ (\tau_1, \tau_2)$.

Typing Contexts	$\Gamma ::=$	$\cdot$	Empty context
	$\mid$	$\Gamma, s{:}\tau$	
	$\mid$	$\Gamma, \alpha{\preceq}\tau{::}\kappa$	
Kinds	$\kappa ::=$	In	Inexact object types
	$\mid$	Ex	Exact object types
	$\mid$	Ty	All types
Types	$\tau ::=$	$\{\!\mid\,\mid\!\}$	Empty object type
	$\mid$	α	Type variable
	$\mid$	$@\tau$	Exact type
	$\mid$	$\tau{\leftarrow\!\!+}\tau$	Extended object type
	$\mid$	$\tau{\Rightarrow}\tau$	Index type
	$\mid$	$\forall\alpha{\preceq}\tau{::}\kappa.\tau$	Polymorphic type
Values	$v ::=$	s	Term variable
	$\mid$	$\mathsf{obj}\ s.\{\!\mid e, \ldots, e \mid\!\}$	Object value
	$\mid$	$\ulcorner n \urcorner$	Index constant ($n \geq 1$)
	$\mid$	$\Lambda\alpha{\preceq}\tau{::}\kappa.e$	Type abstraction
Terms	$e ::=$	v	
	$\mid$	$e{\leftarrow\!\!+}(s){=}e{:}\tau$	Object extension
	$\mid$	$e{\leftarrow}v(s){=}e$	Method override
	$\mid$	$e.v$	Method invocation
	$\mid$	$\mathsf{next\_index}[\tau, \tau]$	Index allocation
	$\mid$	$e[\tau]$	Polymorphic instantiation

Figure 2: Syntax of COI

The types are classified by a kind system containing three constants: Ex, In, and Ty. These classify exact object types, inexact object types, and all types, respectively.

Syntactic values include variables, objects (sequences of methods parameterized by a variable representing `self` or `this`), indices, and type abstractions. Other program terms include object extension, method override, and method invocation. The last two operations require both an expression representing the object to be functionally updated and a value representing the index of the component to be replaced, which could be a variable. The index creation operation $\mathsf{next\_index}[\tau_1, \tau_2]$ returns the index of the first unused index in an object of the $@\tau_1$. Polymorphic instantiation is standard.

2.2 Static and Dynamic Semantics

The full *COI* type system is shown in Appendix A; only a few of the most interesting rules are discussed here.

The kind system is designed specifically for Rule 9,

$$\frac{\Gamma \vdash \tau :: \mathsf{In}}{\Gamma \vdash @\tau :: \mathsf{Ex}}$$

which forbids types of the form $@(@\tau)$ from ever arising. More importantly, if $@\tau$ is valid then τ is the corresponding "inexact" object type. In the presence of type variables and polymorphic instantiation, this is not a purely syntactic restriction.

Rule 10 specifies that only objects of exact types can be extended. (In all following examples, τ_1 will be syntactically of the form $@\tau$.)

$$\frac{\Gamma \vdash \tau_1 :: \mathsf{Ex} \qquad \Gamma \vdash \tau_2 :: \mathsf{Ty}}{\Gamma \vdash \tau_1{\leftarrow\!\!+}\tau_2 :: \mathsf{In}}$$

This restriction ensures that the location of the next component can be predicted just from the type of the object; further, a single prediction will work for every object of this type.

The most important subtyping rule is Rule 18:

$$\frac{\Gamma \vdash \tau_1 :: \mathsf{In} \qquad \Gamma \vdash \tau_2 :: \mathsf{Ty}}{\Gamma \vdash \ ((@\tau_1){\leftarrow\!\!+}\tau_2) <: \tau_1 \ :: \mathsf{In}}$$

A useful specific instance of this rule is $\{\!\mid \sigma_1, \ldots, \sigma_n \mid\!\} \preceq \{\!\mid \sigma_1, \ldots, \sigma_{n-1} \mid\!\}$, i.e., (prefix) width-subtyping for object types. This rule would be unsound if τ_1 were permitted to be exact, as it would allow nontrivial subtypes of exact types.

Because an index of type $\tau_1{\Rightarrow}\tau_2$ can be used both to invoke a method of type τ_2 and to override that method, index types are invariant in the type of the component, as shown in Rule 19:

$$\frac{\Gamma \vdash \ \tau_1' <: \tau_1 \ :: \mathsf{Ty} \qquad \Gamma \vdash \tau :: \mathsf{Ty}}{\Gamma \vdash \ (\tau_1{\Rightarrow}\tau) <: (\tau_1'{\Rightarrow}\tau) \ :: \mathsf{Ty}}$$

The local reduction steps for the dynamic semantics appear in Appendix B. Given the decision to access object components by index, these are straightforward. Primitive steps can be extended to evaluation for full

```
structure Point =
  struct
    type T = {| int, int |}
    val x    = next_index[{| |}][int]
    val get1 = next_index[{|int|}][int]
    fun new (x0:int) = obj s.{| |}
                         ⇷(s)=x0 : int
                         ⇷(s)=s.x : int
    fun eq (o1:T,o2:T) = (o1.x = o2.x)
  end :>
  sig
    type T <: {| |} :: In
    val get1 : T => int
    val new  : int -> @T
    val eq   : T*T -> bool
  end

structure Point2D =
  struct
    type T = @Point.T ⇷ (real,real)
    val x    = next_index[Point.T][real]
    val get2 = next_index[(@Point.T)⇷real][real]
    val get1 = Point.get1
    fun new (x1, x2) = Point.new(x1)
                         ⇷(s) = x2  : real
                         ⇷(s) = s.x : real
  end :>
  sig
    type T <: Point.T :: In
    val get1 : T => int
    val get2 : T => real
    val new  : int*real -> @T
  end
```

Figure 3: Class Encodings with Abstract Types

programs by using evaluation contexts, defined by the grammar:

$$E ::= \bullet \mid E.v \mid E{\leftarrow\!\!+}(s){=}e{:}\tau \mid E{\leftarrow}v(s){=}e.$$

$E[e]$ represents the replacement of the "hole" $\bullet$ in E with the expression e, and a closed program e_1 steps to the program e_2 if $e_1 = E[e_1']$, $e_2 = E[e_2']$, and $e_1' \rightsquigarrow e_2'$ is an instance of a primitive step from Appendix B. This defines the small-step operational semantics of the language [WF91].

Note that the primitive step for $\mathsf{next\_index}[\tau_1, \tau_2]$ depends on the particular type τ_1. This means that either the compiler must be able to determine τ_1 at compile-time — generally possible given incremental or full-program compilation — or some (inexact object) types must be generated and tested at run-time [HM95, Mor95]. From a software engineering point of view it is preferable that a subclass can be typechecked and compiled independently of the implementation details of its superclasses, but a consequence may be that the compiler does not know the superclasses' layout at compile-time.

The type system is appropriate to the dynamic semantics, in that evaluation either terminates with a value or continues indefinitely, but cannot get "stuck":

Proposition 1 (Type Soundness)

1. *If* $\vdash e : \tau$ *and* $e \rightsquigarrow e'$ *then* $\vdash e' : \tau$.
2. *If* $\vdash e : \tau$ *then either* e *is a value or there exists* e' *such that* $e \rightsquigarrow e'$.

2.3 Class Encodings Revisited

Figure 3 shows a revised encoding for the inheritance example, now expressed with COI primitives. Although much of the code looks superficially similar, a method invocation such as `o1.getx` is no longer specifying the method with label `getx` in the object `o1`, but an invocation of the method specified by the value of the variable `getx`.

The type `Point.T` remains partially-abstract, but now we can define `Point2D.T` as a strict subtype. We can also distinguish the type `@Point.T` (classifying objects directly created by the `Point` class via `new`) from the type `Point.T` (classifying objects created either by `Point` or a subclass). Code using these classes could look like:

```
let
  val p1 : @Point.T = Point.new(0)
  val p2 : @Point2D.T = Point2D.new(4,5)
  val n : int  = p2.(Point2D.get1)
  val m : real = p2.(Point2D.get2)
in
  [p1,p2] : Point.T list
end
```

2.4 Encoding Mixins

If classes are SML structures, we can encode mixins as functors; a mixin that adds a private color field along with a getter method is shown in Figure 4.

The mixin could be used as follows:

```
structure CPt =
   struct
      open Point
      open ColorMix(Point)
   end
val cp  : @CPt.T  = CPt.new(3,CPt.Red)
val z   : int = cp.(CPt.getx)
val col : CPt.color = cp.(CPt.getc)
```

3 Language and Encoding Extensions

3.1 Depth Subtyping and Variance Annotations

The type $\tau_1 \Rightarrow \tau_2$ must be invariant in its range type τ_2. However, Abadi and Cardelli [AC96] note that by using *variance annotations* to restrict override or invocation on a method-by-method basis, one can obtain more subtypings. Once could replace index types with types of the form $\tau_1 \Rightarrow^a \tau_2$ with $a \in \{+, 0, -\}$. An index of type $\tau_1 \Rightarrow^+ \tau_2$ can be used to invoke but not override a method; such types can be covariant in τ_2. Similarly, an index of type $\tau_1 \Rightarrow^- \tau_2$ can be used to override but not invoke a method, and such types can be contravariant in τ_2. Finally, indices of type $\tau_1 \Rightarrow^0 \tau_2$ can be used to either override or invoke a method and remain invariant, as before.

3.2 Interfaces

Object types with labeled components in other object calculi could be encoded in COI as existential types, in a manner similar to the object encodings of Pierce and Turner [PT93]. For example, the object type

$$\{\!|\ \mathtt{get1} : \mathtt{int},\ \mathtt{get2} : \mathtt{real}\ |\!\}$$

could be encoded as

$$\exists \alpha \preceq \{\!|\,|\!\}.\ \alpha \times \{\mathtt{get1} : \alpha \Rightarrow \mathtt{int}, \mathtt{get2} : \alpha \Rightarrow \mathtt{real}\}$$

To permit depth subtyping on interfaces (by ruling out code that takes apart this package and overrides the two methods) one could add variance annotations:

$$\exists \alpha \preceq \{\!|\,|\!\}.\ \alpha \times \{\mathtt{get1} : \alpha \Rightarrow^+ \mathtt{int}, \mathtt{get2} : \alpha \Rightarrow^+ \mathtt{int}\}$$

3.3 Renaming Methods

The RS calculus included a renaming operation, which took an object and created a new object with the same components but with different names associated with the components. This is even easier in COI; a subclass can bind an index created by a superclass to a new variable name. This has the advantage that the type of the object itself does not change; subclasses create subtypes regardless of the names of the variables holding indices.

```
functor ColorMix(P : sig
                        type T :: In
                        val new : int -> @T
                     end) =
   struct
      type T   = @P.T ↤↦ (color, color)
      datatype color = Red | Blue
      val c    = next_index[P.T,color]
      val getc = next_index[@P.T↤↦color, color]
      fun new(z0:int, c0:color) =
          P.new(z0) ↤↦ c(s) = c0 : color
                    ↤↦ getc(s) = s.c : color
   end :> sig
             type T <: P.T :: In
             datatype color = Red | Blue
             val getc : T => color
             fun new  : int*color -> @T
          end
```

Figure 4: Encoding of a Mixin which adds Color

4 Related Work

The COI calculus makes use of a number of ideas which have appeared in other sources. In addition to those already mentioned, COI is very reminiscent of the untyped language $\lambda inks$ studied by of Fisher, Reppy, and Riecke [FRR00], though not by design. Their language is intended for relatively low-level compiler representations of object-oriented languages; it may be possible to use the ideas here to obtain a typed variant. Ohori [Oho95] describes an typed intermediate language for compiling polymorphic record calculi (without subsumption) which uses integer indices corresponding to record components; the goals and mechanisms seem very different, however. The C++ language includes pointers-to-member, which are essentially equivalent to the indices here; I am not aware of a formal treatment.

One extension of the RS system handling covariant self-types allowed dictionaries (mappings from names to module components) to be first-class values [RS98]. Later, Vouillon [Vou01] significantly extended the RS system with first-class "views" (named interfaces) to permit encodings of inheritance-based subtyping and binary methods. An index value here might be considered a very primitive form of first-class dictionary or view, describing exactly one component. In both of the earlier systems, however, objects included a "default" dictionary or view, which is absent in COI.

The idea of avoiding name clashes by avoiding labels in objects can also occurs in multimethod approaches to object-oriented programming [DG87, Cha92]. Multimethods — functions which dynamically dispatch based on the types (or classes) of their arguments — are bound to variables, and hence their names can formally alpha-vary as well.

The distinction between exact and inexact object types in COI is somewhat parallel to the distinction between (extensible) prototype and (inextensible) object types of Fisher and Mitchell [FM95]. For example, for soundness both calculi must typecheck method bodies assuming that the variable representing "self" is inextensible. The analogy is not perfect, however; COI does not prevent methods from being overridden in objects with inexact types, and it would not be intrinsically unsound to permit extension for objects with inexact types, so long as the index of the new method were made an output of this process rather than an input. (Typechecking such a modification would become more difficult, however.)

Other approaches have been suggested to achieve by-name subtyping of classes. Fisher and Mitchell [FM98] first define all the classes with exposed interfaces, and then use abstraction after-the-fact to hide everything but the by-name hierarchy from client code using these classes; this may cause difficulties for modularity and separate compilation. League et al. [LST99] get the proper subtyping behavior by encoding the class hierarchy within the types of objects; this requires each class to know the identity of all superclasses and requires object layouts containing extra indirections. Alternatively, Bono et al. [BPSM99] and Fisher and Reppy [FR00] have studied the difficulties in encoding classes with standard extensible objects and have suggested extending object calculi with primitive class mechanisms.

5 Conclusion

By removing all labels from objects, the COI calculus permits encodings of useful class and mixin constructs. These are quite clearly encodings — the facilities of this calculus are much too low-level to want to use directly — but it is interesting to consider the language's suitability for describing lower-level implementations of class-based languages.

The calculus has a number of desirable properties, including the ability to create strict subtypes of partially-abstract types. This removes a principal difficulty of encoding the class inheritance hierarchy as a type hierarchy using standard type abstraction techniques.

References

[Aba94] Martín Abadi. Baby Modula-3 and a theory of objects. *Journal of Functional Programming*, 4(2):249–283, 1994.

[AC96] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[BPSM99] Viviana Bono, Amit J. Patel, Vitaly Shmatikov, and John C. Mitchell. A core calculus of classes and objects. In *Fifteenth Confertence on the Mathematical Foundations of Programming Semantics*, 1999.

[Cha92] Craig Chambers. Object-oriented multimethods in Cecil. In *Proceedings of ECOOP '92 European Conference on Object-Oriented Programming*, volume 615 of *LNCS*, pages 33–56, 1992.

[CM89] Luca Cardelli and John C. Mitchell. Operations on records. In *Proceedings of the Fifth Conference on the Mathematical Foundations of Programming Semantics*. Springer Verlag, 1989.

[Cra98] Karl F. Crary. *Type-Theoretic Methodology for Practical Programming Languages*. PhD thesis, Department of Computer Science, Cornell University, 1998.

[DG87] L. G. DeMichiel and R. P. Gabriel. Common Lisp Object System overview. In Bézivin, Hullot, Cointe, and Lieberman, editors, *Proceedings of ECOOP '87 European Conference On Object-Oriented Programming*, volume 276 of *LNCS*, pages 151–170, June 1987.

[FHM94] Kathleen Fisher, Furio Honsell, and John C. Mitchell. A lambda calculus of objects and method specialization. *Nordic J. Computing*, 1:3–37, 1994. Preliminary version appeared in *Proc. IEEE Symp. on Logic in Computer Science,* 1993, 26–38.

[FM95] Kathleen Fisher and John C. Mitchell. A delegation-based object calculus with subtyping. In *Fundamentals of Computation Theory (FCT'95)*, number 965 in Lecture Notes in Computer Science, pages 42–61. Springer-Verlag, 1995.

[FM98] Kathleen Fisher and John C. Mitchell. On the relationship between classes, objects, and data abstraction. *Theory and Practice of Object Systems*, 4(1):3–25, January 1998.

[FR00] Kathleen Fisher and John Reppy. Extending Moby with inheritance-based subtyping. In *14th European Conference on Object-Oriented Programming*, volume 1850, pages 83–107, 2000.

[FRR00] Kathleen Fisher, John H. Reppy, and Jon G. Riecke. A calculus for compiling and linking classes. In *European Symposium on Programming*, pages 135–149, 2000.

[HM95] Robert Harper and Greg Morrisett. Compiling Polymorphism using Intensional Type Analysis. In *Proc. 22nd ACM Symposium on Principles of Programming Languages (POPL '95)*, pages 130–141, 1995.

[LST99] Christopher League, Zhong Shao, and Valery Trifonov. Representing java classes in a typed intermediate language. In *International Conference on Functional Programming*, pages 183–196, 1999.

[Mor95] Greg Morrisett. *Compiling with Types*. PhD thesis, Carnegie Mellon University School of Computer Science, 1995. Available as CMU Technical Report CMU-CS-95-226.

[Oho95] Atsushi Ohori. A polymorphic record calculus and its compilation. *ACM Transactions on Programming Languages and Systems*, 17(6):844–895, November 1995.

[PT93] Banjamin C. Pierce and David Turner. Object-oriented programming without recursive types. In *Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina*, pages 299–312, 1993.

[RS98] Jon C. Riecke and Christopher A. Stone. Privacy via subsumption. In *Fifth International Workshop on Foundations of Object-Oriented Programming (FOOL 5)*, January 1998.

[Vou01] Jérôme Vouillon. Combining subsumption and binary methods: An object calculus with views. In *Proceedings of the 28th ACM Symposium on Principles of Programming Languages*, pages 290–302, 2001.

[WF91] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. Technical Report TR91-160, Department of Computer Science, Rice University, 1991.

A Full Static Semantics

A.1 Well-Formed Context

$$\frac{}{\cdot \vdash \mathrm{ok}} \tag{1}$$

$$\frac{\Gamma \vdash \tau :: \mathsf{Ty} \qquad s \notin \mathrm{dom}\,\Gamma}{\Gamma, s{:}\tau \vdash \mathrm{ok}} \tag{2}$$

$$\frac{\Gamma \vdash \tau :: \kappa \qquad \alpha \notin \mathrm{dom}\,\Gamma}{\Gamma, \alpha{\preceq}\tau{::}\kappa \vdash \mathrm{ok}} \tag{3}$$

A.2 Subkinding

$$\frac{\Gamma \vdash \text{ok}}{\Gamma \vdash \mathsf{Ex} <: \mathsf{Ty}} \tag{4}$$

$$\frac{\Gamma \vdash \text{ok}}{\Gamma \vdash \mathsf{In} <: \mathsf{Ty}} \tag{5}$$

$$\frac{\Gamma \vdash \text{ok}}{\Gamma \vdash \kappa <: \kappa} \tag{6}$$

A.3 Well-Formed Types

$$\frac{\Gamma \vdash \text{ok}}{\Gamma \vdash \{\!|\,|\!\} :: \mathsf{In}} \tag{7}$$

$$\frac{\Gamma \vdash \text{ok} \qquad \Gamma = \ldots, \alpha \preceq \tau :: \kappa, \ldots}{\Gamma \vdash \alpha :: \kappa} \tag{8}$$

$$\frac{\Gamma \vdash \tau :: \mathsf{In}}{\Gamma \vdash @\tau :: \mathsf{Ex}} \tag{9}$$

$$\frac{\Gamma \vdash \tau_1 :: \mathsf{Ex} \qquad \Gamma \vdash \tau_2 :: \mathsf{Ty}}{\Gamma \vdash \tau_1 \leftarrow\!\!+ \tau_2 :: \mathsf{In}} \tag{10}$$

$$\frac{\Gamma \vdash \tau_1 :: \mathsf{Ty} \qquad \Gamma \vdash \tau_2 :: \mathsf{Ty}}{\Gamma \vdash \tau_1 \Rightarrow \tau_2 :: \mathsf{Ty}} \tag{11}$$

$$\frac{\Gamma \vdash \tau_1 :: \kappa \qquad \Gamma, \alpha \preceq \tau_1 :: \kappa \vdash \tau_2 :: \mathsf{Ty}}{\Gamma \vdash (\forall \alpha \preceq \tau_1 :: \kappa . \tau_2) :: \mathsf{Ty}} \tag{12}$$

$$\frac{\Gamma \vdash \tau :: \kappa_1 \qquad \Gamma \vdash \kappa_1 <: \kappa_2}{\Gamma \vdash \tau :: \kappa_2} \tag{13}$$

A.4 Subtyping

$$\frac{\Gamma \vdash \tau :: \kappa}{\Gamma \vdash \ \tau <: \tau \ :: \kappa} \tag{14}$$

$$\frac{\Gamma \vdash \ \tau_1 <: \tau_2 \ :: \kappa \qquad \Gamma \vdash \ \tau_2 <: \tau_3 \ :: \kappa}{\Gamma \vdash \ \tau_1 <: \tau_3 \ :: \kappa} \tag{15}$$

$$\frac{\Gamma \vdash \text{ok} \qquad \Gamma = \ldots, \alpha \preceq \tau :: \kappa, \ldots}{\Gamma \vdash \ \alpha <: \tau \ :: \kappa} \tag{16}$$

$$\frac{\Gamma \vdash \tau :: \mathsf{In}}{\Gamma \vdash \ @\tau <: \tau \ :: \mathsf{Ty}} \tag{17}$$

$$\frac{\Gamma \vdash \tau_1 :: \mathsf{In} \qquad \Gamma \vdash \tau_2 :: \mathsf{Ty}}{\Gamma \vdash \ ((@\tau_1) \leftarrow\!\!+ \tau_2) <: \tau_1 \ :: \mathsf{In}} \tag{18}$$

$$\frac{\Gamma \vdash \ \tau_1' <: \tau_1 \ :: \mathsf{Ty} \qquad \Gamma \vdash \tau :: \mathsf{Ty}}{\Gamma \vdash \ (\tau_1 \Rightarrow \tau) <: (\tau_1' \Rightarrow \tau) \ :: \mathsf{Ty}} \tag{19}$$

$$\frac{\begin{array}{c}\Gamma \vdash \ \tau_1' <: \tau_1 \ :: \kappa \\ \Gamma, \alpha \preceq \tau_1' :: \kappa \vdash \ \tau_2 <: \tau_2' \ :: \mathsf{Ty} \\ \Gamma, \alpha \preceq \tau_1 :: \kappa \vdash \tau_2 :: \mathsf{Ty}\end{array}}{\Gamma \vdash \ (\forall \alpha \preceq \tau_1 :: \kappa . \tau_2) <: (\forall \alpha \preceq \tau_1' :: \kappa . \tau_2') \ :: \mathsf{Ty}} \tag{20}$$

$$\frac{\Gamma \vdash \ \tau_1 <: \tau_2 \ :: \kappa_1 \qquad \Gamma \vdash \kappa_1 <: \kappa_2}{\Gamma \vdash \ \tau_1 <: \tau_2 \ :: \kappa_2} \tag{21}$$

A.5 Well-Formed Expressions

$$\frac{\Gamma \vdash \text{ok} \qquad s \in \operatorname{dom} \Gamma}{\Gamma \vdash \ s \ : \ \Gamma(s)} \tag{22}$$

$$\frac{\begin{array}{c}\Gamma, s{:}\{\!|\, \tau_1, \ldots, \tau_n \,|\!\} \vdash \ e_1 \ : \ \tau_1 \\ \vdots \\ \Gamma, s{:}\{\!|\, \tau_1, \ldots, \tau_n \,|\!\} \vdash \ e_n \ : \ \tau_n\end{array}}{\begin{array}{r}\Gamma \vdash \ \mathsf{obj}\ s.\{\!|e_1{:}\tau_1, \ldots, e_n{:}\tau_n|\!\} \ : \\ @\{\!|\, \tau_1, \ldots, \tau_n \,|\!\}\end{array}} \tag{23}$$

$$\frac{\Gamma \vdash \ e \ : \ \tau_1 \qquad \Gamma \vdash \ v \ : \ \tau_1 \Rightarrow \tau}{\Gamma \vdash \ e.v \ : \ \tau} \tag{24}$$

$$\frac{\begin{array}{c}\Gamma \vdash \ e_1 \ : \ \tau_1 \\ \Gamma \vdash \ v \ : \ \tau_1 \Rightarrow \tau_3 \\ \Gamma, s{:}\tau_1 \vdash \ e_3 \ : \ \tau_3\end{array}}{\Gamma \vdash \ e_1 \leftarrow v(s) = e_3 \ : \ \tau_1} \tag{25}$$

$$\frac{\begin{array}{c}\Gamma \vdash \ e_1 \ : \ @\tau_1 \\ \Gamma, s{:}(@\tau_1) \leftarrow\!\!+ \tau_2 \vdash \ e_2 \ : \ \tau_2\end{array}}{\Gamma \vdash \ e_1 \leftarrow\!\!+ (s) = e_2{:}\tau_2 \ : \ @((@\tau_1) \leftarrow\!\!+ \tau_2)} \tag{26}$$

$$\frac{\Gamma \vdash \tau_1 :: \mathsf{In} \qquad \Gamma \vdash \tau_2 :: \mathsf{Ty}}{\Gamma \vdash \ \mathsf{next\_index}[\tau_1, \tau_2] \ : \ ((@\tau_1) \leftarrow\!\!+ \tau_2) \Rightarrow \tau_2} \tag{27}$$

$$\frac{\Gamma \vdash \tau_1 :: \mathsf{Ty} \quad \cdots \quad \Gamma \vdash \tau_n :: \mathsf{Ty} \qquad 1 \le k \le n}{\Gamma \vdash \ \ulcorner k \urcorner \ : \ \{\!|\, \tau_1, \ldots, \tau_n \,|\!\} \Rightarrow \tau_k} \tag{28}$$

$$\frac{\Gamma, \alpha \preceq \tau_1 :: \kappa \vdash \ e \ : \ \tau_2}{\Gamma \vdash \ \Lambda \alpha \preceq \tau_1 :: \kappa . e \ : \ \forall \alpha \preceq \tau_1 :: \kappa . \tau_2} \tag{29}$$

$$\frac{\Gamma \vdash \ e \ : \ (\forall \alpha \preceq \tau_1 :: \kappa . \tau_2) \qquad \Gamma \vdash \ \tau <: \tau_1 \ :: \kappa}{\Gamma \vdash \ e[\tau] \ : \ \tau_2[\alpha \mapsto \tau]} \tag{30}$$

$$\frac{\Gamma \vdash \ e \ : \ \tau_1 \qquad \Gamma \vdash \ \tau_1 <: \tau_2 \ :: \mathsf{Ty}}{\Gamma \vdash \ e \ : \ \tau_2} \tag{31}$$

B Dynamic Semantics

$$\begin{array}{c}(\mathsf{obj}\ s.\{\!|e_1{:}\tau_1, \ldots, e_n{:}\tau_n|\!\}).\ulcorner k \urcorner \\ \rightsquigarrow \\ e_k[s \mapsto \mathsf{obj}\ s.\{\!|e_1{:}\tau_1, \ldots, e_n{:}\tau_n|\!\}] \qquad (\text{if } k \in 1..n)\end{array}$$

$$\begin{array}{c}(\mathsf{obj}\ s.\{\!|e_1{:}\tau_1, \ldots, e_n{:}\tau_n|\!\}) \leftarrow\!\!+ (s) = e_{n+1}{:}\tau_{n+1} \\ \rightsquigarrow \\ \mathsf{obj}\ s.\{\!|e_1{:}\tau_1, \ldots, e_{n+1}{:}\tau_{n+1}|\!\}\end{array}$$

$$\begin{array}{c}(\mathsf{obj}\ s.\{\!|e_1{:}\tau_1, \ldots, e_n{:}\tau_n|\!\}) \leftarrow \ulcorner k \urcorner (s) = e_k' \\ \rightsquigarrow \\ \mathsf{obj}\ s.\{\!|e_1{:}\tau_1, \ldots, e_{k-1}{:}\tau_{k-1}, e_k'{:}\tau_k, e_{k+1}{:}\tau_{k+1}, \ldots, e_n{:}\tau_n|\!\}\end{array}$$

$$\mathsf{next\_index}[\{\!|\, \sigma_1, \ldots, \sigma_n \,|\!\}, \tau_2] \ \rightsquigarrow \ \ulcorner n+1 \urcorner$$

$$(\Lambda \alpha \preceq \tau :: \kappa . e)[\tau] \ \rightsquigarrow \ e[\alpha \mapsto \tau]$$