Extensible Objects Without Labels

CHRISTOPHER A. STONE
Harvey Mudd College

Typed object calculi that permit adding new methods to existing objects must address the problem of name clashes: what happens if a new method is added to an object already having one with the same name but a different type? Most systems statically forbid such clashes by restricting the allowable subtypings. In contrast, by reconsidering the runtime meaning of object extension, the object calculus studied in the author's previous work with Jon Riecke allowed any object to be soundly extended with any method of any name, with unrestricted width subtyping. That language permitted a simple encoding of classes as object-generators. Because of width subtyping, subclasses could be typechecked and compiled with little knowledge of the class hierarchy and without any information about superclasses' private components; this made derived classes more robust to changes in the implementations of base classes. However, the system was not well-suited for encoding mixins or by-name subtyping of objects.

This paper addresses those deficiencies by presenting the Calculus of Objects and Indices (COI), a lower-level typed object calculus in which extensible objects are more analogous to tuples than to records. An object is simply a finite sequence of unnamed components referenced by their index in the sequence. Names are then re-introduced by allowing these indices to be first-class values (analogous to pointers to members in C++) that can be bound to variables. Since variables — unlike record labels — freely alpha-vary, difficulties caused by statically undetectable name clashes disappear.

By combining COI objects with standard type-theoretic mechanisms, one can encode mixins and classes having the by-name subtyping of languages like C++ or Java but with the robustness of the object-generator encodings. Using records, more standard extensible objects with named components can also be encoded.

Categories and Subject Descriptors: []:

General Terms: Languages

Additional Key Words and Phrases: Object Calculi, Extensible Objects

1. INTRODUCTION

Early work on the foundations of object-oriented languages treated objects as particular uses of records, e.g., "Object-oriented programming is based on record structures (called *objects*) intended as named collection of values (*attributes*) and functions (*methods*)" [Cardelli and Mitchell 1989]. Although it has more recently become common to study systems in which objects are primitive [Abadi and Cardelli 1996; Fisher et al. 1994], such objects typically remain syntactically record-like: each component is identified by and accessed with a unique label, the name of the

Author's address: Christopher A. Stone, Department of Computer Science, Harvey Mudd College, 1250 N. Dartmouth Ave., Claremont CA 91101.

field or method.

This can cause difficulties in typed systems that permit new fields or methods to be added to existing objects. Consider, for example, a simple object with two methods: a component `n` containing an integer constant, and a accessor method `getn` for that component:

$$\mathsf{obj}\ \mathit{self}.\{|\texttt{n = 17 : int, getn = }\mathit{self}\texttt{.n : int}|\}.$$

Here, the variable *self* is a bound variable representing the entire object (sometimes called `this` in object-oriented languages), which can be used in method code. (As usual, for simplicity fields are modeled as constant methods not referencing *self*.) Since both components return the integer `17` when invoked — `n` directly and `getn` indirectly — the object naturally has the (structural) object type

$$\{|\,\texttt{n : int, getn : int}\,|\}.$$

Under the usual rules of width subtyping for objects, this object also has the type

$$\{|\,\texttt{getn : int}\,|\},$$

which, though less specific, is arguably more appropriate if the `n` field is expected to be accessed only via the accessor method `getn`.

Suppose this object is now extended with an `n` component returning the boolean value `true`. If all that is known statically about the object is that it has type $\{|\,\texttt{getn:int}\,|\}$, there is no apparent reason to prohibit such an extension. What should the result be? Overriding the pre-existing `n` field to obtain

$$\mathsf{obj}\ \mathit{self}.\{|\texttt{n = true : bool, getn = }\mathit{self}\texttt{.n : int}|\}$$

would be unsound because this change causes the integer method `getn` to return a boolean, a type error. However, record-like structures cannot have two components with the same label:

$$\mathsf{obj}\ \mathit{self}.\{|\texttt{n = 17 : int, getn = }\mathit{self}\texttt{.n : int, n = true : bool}|\}.$$

because references to the `n` component become ambiguous.

It might appear from this example as though the integer `n` component could automatically be renamed out of the way when the boolean `n` component is added, yielding an object such as

$$\mathsf{obj}\ \mathit{self}.\{|\texttt{n' = 17 : int, getn = }\mathit{self}\texttt{.n' : int, n = true : bool}|\}$$

but this does not work in general. Labels do not have delimited scopes and cannot alpha-vary as variables do. For example, given a function

$$\texttt{project\_out\_n} : \{|\,\texttt{n : int}\,|\} \to \texttt{int}$$

that takes any object with an integer `n` component and returns the value of that component, the original object could have been written equivalently as

$$\mathsf{obj}\ \mathit{self}.\{|\texttt{n = 17 : int, getn = project\_out\_n(}\mathit{self}\texttt{) : int}|\}.$$

In this case renaming the `n` component no longer works; the result would be

$$\begin{array}{l}\mathsf{obj}\ \mathit{self}.\{|\texttt{n' = 17 : int, n = true : bool,}\\ \qquad\texttt{getn = project\_out\_n(}\mathit{self}\texttt{) : int}|\}\end{array}$$

where `getn` again erroneously returns a boolean.

Of course the addition of the second `n` field could simply be detected and reported as a run-time error, but if type systems can guarantee that code never invokes a non-existent method at run time, it seems reasonable to expect object extension to be statically checked as well.

Most systems of extensible objects forbid such problematic extensions by limiting subtyping [Fisher and Mitchell 1995; Liquori 1997]. In the above example, the original object would not be permitted to be given the type $\{\!|\, \texttt{getn : int} \,|\!\}$ without ruling out all further extension; there is always information about all components when objects are extended and so clashes are always preventable. However, this forces the names and/or types of *private* methods — never intended to be externally accessed — to appear in the types of objects as long as extension (inheritance) is possible, violating information-hiding principles.

In contrast, the author's work with Jon Riecke [1998; 2002] allowed arbitrary width subtyping and unrestricted object extension by modifying the representation of objects in the semantics. The key innovation of that calculus (hereafter referred to as RS) was to augment object values with "dictionaries", explicit indirections giving names to corresponding object components. In the above example, the original object could be represented as:

$$\texttt{obj}\ \textit{self}.\{\!|\texttt{17 : int, } \textit{self}.\texttt{1 : int}|\!\}_{[\texttt{n}\mapsto 1, \texttt{getn}\mapsto 2]}.$$

The dictionary $[\texttt{n} \mapsto 1, \texttt{getn} \mapsto 2]$ attached to this object specifies that externally invoking the `n` method of this object will access the first component, while externally invoking the `getn` method will run the code of the second component. Importantly, methods can refer to each other (so-called "self-inflicted" method invocations) directly by their unique index, rather than using the object's dictionary.[1]

The RS syntax for adding a new method is $e_1 \leftarrow\!\!+\, l(s) = e_2 : \tau_2$, which denotes the result of taking the object e_1 and adding a new method named l whose code e_2 of type τ_2 can refer to the entire object via the bound variable s. We can take this last object above and add a new boolean `n` component

$$(\ldots) \leftarrow\!\!+\, \texttt{n}(s) = \texttt{true} : \texttt{bool}$$

to obtain the object

$$\texttt{obj}\ \textit{self}.\{\!|\texttt{17 : int, } \textit{self}.\texttt{1 : int, true : bool}|\!\}_{[\texttt{n}\mapsto 3, \texttt{getn}\mapsto 2]}.$$

The dictionary $[\texttt{n} \mapsto 3, \texttt{getn} \mapsto 2]$ now routes external invocations of `n` to the new boolean component (the original `n` has been *shadowed*), but importantly the behavior of all existing methods — `getn` in particular — remains unchanged. The principal type of this object would be

$$\{\!|\, \texttt{n : bool, getn : int} \,|\!\},$$

which exposes only the types of the two externally-accessible components. In contrast to other calculi [Liquori 1997; Di Gianantonio et al. 1998], RS object extension

[1] In actuality, methods in RS often perform self-inflicted invocations via the dictionary in place when the method code was added (rather than the dictionary at the point when the method was invoked). In cases where that original dictionary is known, the lookup can be statically reduced to the appropriate index.

will never replace the code of an existing method or change the behavior of existing methods. At most a pre-existing method may become externally shadowed as in this example.

An operation on values is said to be *type-preserving* if, whenever the input has type τ, then the output also has type τ. (In the presence of subtyping and subsumption this does not require the output to have the same *principal* type as the input; the output could have a strictly more precise type.) It is clear from the above example that RS object extension is not type-preserving, as the original object has type {| `n : int, getn : int` |} while the extended object has the incomparable type {| `n : bool, getn : int` |}.[2]

The fact that object extension in RS is not type-preserving in general limits the flexibility of the system, especially in the presence of abstract types and type variables. The remainder of this section considers two specific issues: encoding class types with by-name subtyping, and encoding mixins.

1.1 RS Limitation: Class Types

Primitive objects have often been used to study and model classes in conventional object-oriented languages, decomposing relatively complex classes into uses of more primitive type-theoretic constructs. One difficulty, however, is faithfully modeling the type hierarchies found in class-based languages. Most formal studies of primitive objects and subtyping assume structural typing, where the type of an object is determined only by the types of its components. In contrast, classes in C++ or Java act as types with a by-name subtyping hierarchy mirroring the inheritance graph.

One of the most elegant approaches to modeling classes treats them simply as object generators. Specifically, each class is encoded as a module with at least two components: a type `T` representing the type of objects of that class, and a function `new` to construct objects of type `T`.

Figure 1 shows an encoding of a class `Point` and a subclass `Point2D` built using RS objects and a type-theoretic core in the syntax of Standard ML (SML) [Milner et al. 1997]. (SML is used here primarily because it supports the creation of abstract types through its module system.)

The signatures `POINT` and `POINT2D` are module interfaces representing the encodings of a classes for 1-dimensional points and 2-dimensional points respectively. According to `POINT`, objects of the base class have (at least) a `get1` method returning an integer, while `POINT2D` specifies that objects of the subclass have (at least) the methods `get1` and `get2`. The constructor function for 1-dimensional points takes an integer argument, while the constructor function for 2-dimensional points is specified as requiring both an integer and a real. Further, the interface for 1-dimensional points requires an implementation also supply a "friend function" `eq` that works on pairs of 1-dimensional points.

---

[2] Extension is type-preserving in some cases, of course. Had the new component been added with the name `b`, for example, the result would be an object of type

{| `n : int, getn : int, b : bool` |},

a subtype of the original object's type.

```
signature POINT   = sig
                      type T = {| get1 : int |}
                      val new : int -> T
                      val eq  : T*T -> bool
                    end

signature POINT2D = sig
                      type T = {| get1 : int, get2 : real |}
                      val new : int*real -> T
                    end


structure Point :> POINT =
  struct
    type T = {| get1 : int |}
    fun new(x1:int):T = (obj s.{| |} )
                          ←+ x(s) = x1 : int
                          ←+ get1(s) = s.x : int
    fun eq (o1:T,o2:T):bool = (o1.get1 = o2.get1)
  end

structure Point2D :> POINT2D =
  struct
    type T = {| get1 : int, get2 : real |}
    fun new (x1:int, x2:real):T = Point.new(x1)
                                    ←+ x(s) = x2 : real
                                    ←+ get2(s) = s.x : real
  end
```

Fig. 1. Class Encodings using RS Objects

Implementations are the `Point` and `Point2D` modules. The SML syntax

```
structure Point :> POINT = struct ... end
```

defines the `Point` module to contain all the definitions between `struct` and `end`, and then hides any information about this module that does not appear explicitly in the `POINT` interface. For example, the type `Point.T` is a synonym for the type {| `get1:int` |} because this fact appears in the interface `POINT`. In contrast, although the code for the function `Point.new` clearly creates objects of size two by taking the empty object and extending it twice, the `POINT` interface merely guarantees that the resulting object satisfies the supertype `Point.T`, i.e., only that it contains `get1`. The type checker therefore will reject any attempt to access the `x` field of an object returned by `Point.new`, effectively making the `x` field completely private.

The `Point2D` class defines its own object type `Point2D.T`, and creates two-dimensional points by inheritance; the function `Point2D.new` creates objects by first creating an object of the superclass and then extending it with two new components. The result at run time will be an object with four components, but statically only the two accessor methods are accessible because only these are guaranteed to exist by `POINT2D`. Although `Point.new` and `Point2D.new` independently add components named `x` and these components have different types, the use of dic-

tionaries ensures that the two fields nevertheless remain distinct and are by their respective accessors.

This encoding of classes uses purely structural subtyping. Two-dimensional points can be used as one-dimensional points not because of inheritance, but because the definitions of the types `Point.T` and `Point2D.T` exposed in the interfaces are in a subtyping relationship. The type system cannot distinguish an object created by `Point.new` from any other random object having a `get1` method returning an integer. Consequently, 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 so cannot directly access the `x` fields of its arguments; there is no guarantee that such fields even exist. Conversely, if the arguments of `eq` were required to have `x` fields, then the type system would prevent the function from being used with objects created by `Point.new` — the `POINT` interface does not guarantee that all such objects will have `x` fields[3].

A natural type-theoretic fix is to make the type `Point.T` *partially abstract*. Instead of specifying that the type of objects created by `Point.new` satisfies the structural type {| `get1: int` |}, we can say that `Point.T` is some unknown type reflecting the actual implementation of one-dimensional points, and that this unknown type is guaranteed to be a *subtype* of the interface type {| `get1: int` |}. Thus, by subsumption, we can still invoke the `get1` method in an object created by the `Point` class.

This is essentially the approach taken in Modula-3 [Nelson 1991], and although SML does not include partially-abstract types in interfaces it would be a natural extension in the presence of subtyping [Crary 1998]. In this case, the `POINT` interface and `Point` module can be rewritten as shown in Figure 2. `Point.T` is now the type of the object's internal representation (including the private `x` field) rather than an interface. The code for `Point.new` is unchanged, but now the code for `Point.eq` requires its arguments satisfy the implementation type and hence the code can directly compare the contents of the `x` fields.

Clients of this class can refer to the abstract type `Point.T` when an object of this class is required, and can refer to its supertype {| `get1:int` |} when only the public interface matters. Abstraction ensures that `eq` will be applied only to objects created by this class, and hence it is sound for `eq` to assume its arguments have `x` fields. The `x` field remains private (hidden and inaccessible outside the `Point` module).

Unfortunately, as observed by Fisher and Reppy [2000] this change prevents useful inheritance. The definition of `Point.T` is hidden by the `POINT` interface, and hence there is no way to define a strict subtype `Point2D.T`. As extension is not always type-preserving, there is no way to guarantee that the objects created by `Point2D.new` (which calls `Point.new` and extends the result), still have the

[3] `Point.new` could randomly choose whether or not to return an object containing an `x` field without violating the `POINT` interface. Changing the `POINT` interface to require such objects contain `x` would leak private implementation details into the interface of the class, defeating the whole point (as it were) of having RS objects in the first place.

```
signature POINT = sig
                    type T ⪯ {| get1 : int |}  (* Partially abstract *)
                    val new : int -> T
                    val eq  : T*T -> bool
                  end =

structure Point :> POINT =
  struct
    type T = {| x : int, get1 : int |}
    fun new(x1:int):T = (obj s.{| |})
                            ←+ x(s) = x1 : int
                            ←+ get1(s) = s.x : int
    fun eq (o1:T,o2:T):bool = (o1.x = o2.x)
  end
```

Fig. 2. Revised RS Encoding for `Point`

implementation type `Point.T`. In fact, the run-time object values created by the `Point2D` class in Figure 1 have principal type

$$\{|\ \texttt{x : real, get1 : int, get2 : real}\ |\}$$

(because there are four components, with the integer `x` having been shadowed in the dictionary) and this is not a subtype of `Point`'s implementation type

$$\{|\ \texttt{x : int, get1 : int}\ |\}.$$

Consequently, objects created by `Point2D.new` must not be passed to the `Point.eq` function in Figure 2; that code is expecting to use integer equality, but accessing the `x` fields of `Point2D` objects yields reals.

Modula-3 does permit the creation of subtypes of such partially abstract object types exactly as needed here, but, this does not seem to have been well-formalized. Although the work of Abadi [1994] on Baby Modula-3, for example, includes an operator for extending objects, this operation is artificially restricted to extending only object values where all components are syntactically apparent and hence where name clashes can be statically detected and forbidden.

1.2 RS Limitation: Mixins

The RS type system also cannot give the most useful types to code which extends objects. This makes the language unsuitable for encoding constructs such as mixins and parameterized classes. Ideally, one would like to be able to write code to extend an arbitrary object with both public and private (hidden) components, and to track this extension in the type system. However, again because of the potential for name clashes this is not possible even using dictionaries.

In RS 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 very simple mixin [Bracha and Cook 1990; Flatt et al. 1998]. However, the type of this function would be

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

That is, `colorize` can be applied to any object at all by subsumption, but it guarantees only that the resulting object has a `getc` method. Because this function

extends its argument, and because extension in RS never replaces existing components, we know the object returned still contains all the components present in the argument. The type does not express this fact.

Simply adding bounded polymorphism would not help, as the best type for such a function would be

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

which again does not express how the contents of the object returned depend on the argument.

Cardelli and Mitchell [1989] addressed a similar 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 object type constructor $\tau{\leftarrow\!\!+}l{:}\sigma$ (the type of objects created by taking an object of type τ and adding a new component l of type σ) to RS, `colorize` could be written to have the type

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

which very accurately describes the behavior of the code in question. Unfortunately, such types are not very flexible. Since object extension is not type-preserving, the type $\tau{\leftarrow\!\!+}l{:}\sigma$ cannot be defined as a subtype of τ; width subtyping cannot extend to "forgetting" these extensions. As a consequence, a variant definition of `colorize` which adds both a field `c` and a accessor method `getc` could reasonably be given the very precise type

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

or the less precise type

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

or, if the `c` field should be private and externally hidden, even the type

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

However, the type

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

that both specifies the dependency but omits mention of the private component `c` would be unsound. This last type implies that an argument having a component `c` yields an output with a component `c` of the same type, whereas the assumed code would *shadow* any pre-existing `c` component.

Thus, although types of the form $\tau{\leftarrow\!\!+}l{:}\sigma$ would permit more precise typings for mixins encoded as object-extending functions (or functors, when classes are structures), components added by these mixins cannot be made hidden and inaccessible by subsumption, thus losing one of the greatest benefits of the RS system.

The remainder of the paper addresses the limitations of the RS system by moving still further from tradition of record-like objects. Section 2 introduces an alternative calculus COI in which objects lack labels (and dictionaries) entirely. Section 3 then shows how the above limitations of RS can be overcome using COI. Section 4 extends the language to include variance annotations, while Section 5 sketches how to encode labeled RS objects into COI. Finally, Section 6 concludes.

Typing Contexts	$\Gamma ::=$	$\cdot$	Empty context
		$\mid \Gamma, s{:}\tau$	Term variable specification
		$\mid \Gamma, \alpha \preceq \tau :: \kappa$	Type variable specification
Types	$\tau, \sigma ::=$	$\{\!\|\,\|\!\}$	Empty object type
		$\mid \alpha \mid \beta \mid \cdots$	Type variable
		$\mid @\tau$	Exact type
		$\mid \tau \leftarrow\!\!+ \tau$	Extended object type
		$\mid \tau \Rightarrow \tau$	Index type
		$\mid \exists \alpha \preceq \tau :: \kappa . \tau$	Package type
Kinds	$\kappa ::=$	In	Inexact object types
		$\mid \mathsf{Ex}$	Exact object types
		$\mid \mathsf{Ty}$	All types
Values	$v ::=$	$s \mid x \mid \ldots$	Term variable
		$\mid \mathsf{obj}\ s.\{\!\|e, \ldots, e\|\!\}$	Object value
		$\mid \ulcorner n \urcorner$	Index constant ($n \geq 1$)
		$\mid \mathsf{pack}\ \tau\ \mathsf{and}\ e\ \mathsf{as}\ \tau$	Existential package
Terms	$e ::=$	v	
		$\mid e \leftarrow\!\!+ (e){=}e{:}\tau$	Object extension
		$\mid e \leftarrow e(s){=}e$	Method override
		$\mid e.e$	Method invocation
		$\mid \mathsf{next\_index}(\tau)$	Index allocation
		$\mid \mathsf{open}\ e\ \mathsf{as}\ \alpha\ \mathsf{and}\ x\ \mathsf{in}\ e$	Unpacking existential

Fig. 3. Syntax of COI

2. THE COI SYSTEM

The Calculus of Objects and Indices (COI) addresses the limitations of RS 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 index numbers are made first-class values, analogous to pointers to members in C++ [Stroustrup 1997].

This design has two very useful consequences. First, adding a new component at the end of an object's sequence cannot affect pre-existing methods in any way, not even by shadowing them. Secondly, since variables — unlike labels — can freely alpha-vary, index values can be bound to variables that can be used to refer to method components without any possibility of unsound name clashes.

2.1 Syntax

The syntax of the COI system system is shown in Figure 3. The simplest type is $\{\!\|\,\|\!\}$, the type of the empty object and, by subsumption, of every object. The type of an object of τ extended with a new component of type σ is written $\tau \leftarrow\!\!+ \sigma$. (This is simpler than the syntax used in Section 1.2 because object components no longer have labels.) Index values have types of the form $\tau \Rightarrow \sigma$; this type classifies offsets which access a component of type σ within an object of type τ. The exact type $@\tau$ classifies objects whose principal (most-specific) type is exactly τ rather than some subtype of τ; extending an object has predictable results (i.e., determining at what offset the new method will appear) only if we know the exact type of the original object. Finally, the type system includes type variables and bounded

existentials [Pierce 2002]; these are not part of the core object calculus but are included only to represent the interaction with type abstraction. Specifically, the type $\exists\alpha{\preceq}\tau_1{::}\kappa.\tau_2$ classifies packages containing a type σ that is a subtype of τ_1 and of kind κ, together with a term of type $\tau_2\big[\sigma/\alpha\big]$.

Types of the form $\{\!|\,|\!\}$ or $\tau{\leftarrow\!\!+}\sigma$ are referred to as *inexact* object types; any object with such a type has *at least* the components mentioned, but possibly more. Conversely, types of type form $@\tau$ are said to be *exact* object types; an object of type $@\tau$ has exactly those components mentioned in the corresponding inexact object type τ, and no more.

By this definition, extending an exact type yields an inexact type; the type $(@\tau){\leftarrow\!\!+}\sigma$ classifies objects that were created by taking an object of exactly type τ, adding a new component of type σ, and possibly further extensions. In contrast, the exact type $@((@\tau){\leftarrow\!\!+}\sigma)$ specifies that there were no further additions after the σ component was added.

It is sometimes important to know whether a given abstract type (or type variable) is exact or not, and so 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.

Two shorthands are useful for writing COI types. More familiar object types — though still with unlabeled components — can be expressed by repeatedly extending the empty object type. Such types are defined recursively as:

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

According to the typing rules in Section 2.3, this classifies all objects beginning with exactly $n-1$ components having types τ_1 through τ_{n-1} in order, immediately followed by an component of type τ_n. This is not an exact type, however, because it leaves open the possibility that there are other components as well. Specifying an objects that have *only* these n components requires the exact type $@\{\!|\,\tau_1,\ldots,\tau_n\,|\!\}$.

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 extend an object type with several components at once:

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

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

At the term level, the syntactic values include variables, objects (sequences of methods parameterized by a variable representing the enclosing object), indices, and existentially-typed packages. Other program terms include object extension, method override, and method invocation. The last two operations require both an expression yielding an object, and an expression (rather than the usual label) yielding the component of interest. The index creation operator $\mathsf{next\_index}(\tau)$ returns the index of the first unused index in an object of the exact type τ. Finally, the elimination form for existentials is standard [Pierce 2002]; a value of an existential type is a pair containing a type and a term, and the open construct binds these two components to local variables for the purposes of evaluating an expression e.

$$
\begin{array}{rcl}
(\mathsf{obj}\ s.\{\!|e_1{:}\tau_1,\ldots,e_n{:}\tau_n|\!\}).\ulcorner k\urcorner & \rightsquigarrow & e_k\big[\mathsf{obj}\ s.\{\!|e_1{:}\tau_1,\ldots,e_n{:}\tau_n|\!\}/s\big] \\
(\mathsf{obj}\ s.\{\!|e_1{:}\tau_1,\ldots,e_n{:}\tau_n|\!\})\leftarrow\!\!+(s){=}e_{n+1}{:}\tau_{n+1} & \rightsquigarrow & \mathsf{obj}\ s.\{\!|e_1{:}\tau_1,\ldots,e_n{:}\tau_n,e_{n+1}{:}\tau_{n+1}|\!\} \\
(\mathsf{obj}\ s.\{\!|e_1{:}\tau_1,\ldots,e_n{:}\tau_n|\!\})\leftarrow\ulcorner k\urcorner(s){=}e'_k & \rightsquigarrow & \mathsf{obj}\ s.\{\!|e_1{:}\tau_1,\ldots,e_{k-1}{:}\tau_{k-1},e'_k{:}\tau_k,\ldots,e_n{:}\tau_n|\!\} \\
\mathsf{next\_index}(@\{\!|\,\sigma_1,\ldots,\sigma_n\,|\!\}) & \rightsquigarrow & \ulcorner n{+}1\urcorner \\
\mathsf{open}\ (\mathsf{pack}\ \tau\ \mathsf{and}\ v\ \mathsf{as}\ \sigma)\ \mathsf{as}\ \alpha\ \mathsf{and}\ x\ \mathsf{in}\ e & \rightsquigarrow & e\big[\tau/\alpha\big]\big[v/x\big]
\end{array}
$$

Fig. 4. Primitive Reduction Steps for COI

$$
\begin{array}{rcl}
\mathcal{E} & ::= & \bullet \\
& | & \mathcal{E}.e \\
& | & v.\mathcal{E} \\
& | & \mathcal{E}\leftarrow\!\!+(s){=}e{:}\tau \\
& | & \mathcal{E}\leftarrow e_2(s){=}e_3 \\
& | & v_1\leftarrow\mathcal{E}(s){=}e_3 \\
& | & \mathsf{pack}\ \tau\ \mathsf{and}\ \mathcal{E}\ \mathsf{as}\ \sigma
\end{array}
$$

Fig. 5. Evaluation Contexts

2.2 Dynamic Semantics

The primitive reduction steps for the dynamic semantics appear in Figure 4. Given the decision to access object components by run-time index values, these are largely straightforward modifications of the conventional rules.

The only completely unfamiliar rule should be that for $\mathsf{next\_index}(\tau)$, which computes index values of components at run-time. Because this evaluation step depends on the particular type τ, the semantics as presented here does not support a type-erasure interpretation. (Section 6.2 sketches a slightly more complex formulation that eliminates the dependency of the dynamic semantics on types.)

The primitive steps are extended to evaluation for full programs — arbitrary closed expressions — by using the evaluation contexts defined by the grammar in Figure 5. If $\mathcal{E}$ is an evaluation context (An expression with a "hole", written $\bullet$) then $\mathcal{E}[e]$ denotes the term resulting by replacing the hole by the expression e. Following Wright and Felleisen [1991], the small-step evaluation relation is then extended from primitive steps to programs by specifying that $e_1 \rightsquigarrow e_2$ if there exist $\mathcal{E}$, e'_1, and e'_2 such that $e_1 = \mathcal{E}[e'_1]$, $e_2 = \mathcal{E}[e'_2]$, and $e'_1 \rightsquigarrow e'_2$ is an instance of a primitive step from Figure 4.

For example, one can evaluate code to build an object with a field and an accessor

and to invoke the latter method as follows:

$$\begin{array}{rl} & \big(((\mathsf{obj}\ s.\{\!|\}) \leftarrow\!\!+ (s'){=}17{:}\mathtt{int}) \leftarrow\!\!+ (s''){=}(s''.\ulcorner 1\urcorner){:}\mathtt{int}\big).\ulcorner 2\urcorner \\ \leadsto & \big((\mathsf{obj}\ s.\{\!|17{:}\mathtt{int}|\!\}) \leftarrow\!\!+ (s''){=}(s''.\ulcorner 1\urcorner){:}\mathtt{int}\big).\ulcorner 2\urcorner \\ \leadsto & (\mathsf{obj}\ s.\{\!|17{:}\mathtt{int}, (s.\ulcorner 1\urcorner){:}\mathtt{int}|\!\}).\ulcorner 2\urcorner \\ \leadsto & (s.\ulcorner 1\urcorner)\big[\mathsf{obj}\ s.\{\!|17{:}\mathtt{int}, (s.\ulcorner 1\urcorner){:}\mathtt{int}|\!\}/s\big] \\ = & (\mathsf{obj}\ s.\{\!|17{:}\mathtt{int}, (s.\ulcorner 1\urcorner){:}\mathtt{int}|\!\}).\ulcorner 1\urcorner \\ \leadsto & 17. \end{array}$$

2.3 Static Semantics

The remaining part of the definition of COI is the typing rules, or static semantics. The entire set of rules are shown in Figures 6, 7, and 8; only the most interesting will be discussed further.

2.3.1 *Well-Formed Types.* The kind system, which distinguishes exact object types from inexact object types, is used in Rule 6 to forbid types of the form $@(@\tau)$ from ever arising. Consequently, if $@\tau$ is well-formed then τ is always the corresponding inexact object type.

Rule 7 specifies that only objects with exact types can be extended. This restriction ensures that the location of the next component can be predicted just from the type of the object, so that extending many objects of the same type requires computing the index of the new method only once.

It would not be unsound to allow the extension of arbitrary objects, but doing so does not appear useful. COI has no way to determine the right index for accessing the new method, as extending two objects with the same inexact type $\{\!|\ |\!\}$ could add components at completely different offsets. (It is likely that one could, for example, modify object extension to return both the extended object and the index of the new component. The overhead of tracking in the type system that this particular new index is usable with *only* this particular new object seems significant. Further, that level of flexibility is not needed to overcome the limitations of RS discussed above.)

2.3.2 *Subtyping.* Exact types are subtypes of the corresponding inexact type, according to Rule 17; the premise ensures that the exact type $@\tau$ is well-formed.

The type $(@\tau_1) \leftarrow\!\!+ \tau_2$ classifies objects having at least $n{+}1$ components, with the types of the first n being specified by the inexact object type τ_1, and the $n{+}1^{st}$ having type τ_2. It is clear that such objects do not also have type $@\tau_1$, which classifies objects having exactly n components; thus in general $\sigma_1 \leftarrow\!\!+ \sigma_2$ is not a subtype of σ_1. However, objects of type $(@\tau_1) \leftarrow\!\!+ \tau_2$ do have at least the n components specified in the inexact type τ_1, and so $(@\tau_1) \leftarrow\!\!+ \tau_2$ can be a subtype of τ_1. (This is the point where the kind structure becomes important. Although $@(@\tau)$ may be semantically reasonable as a redundant way of representing $@\tau$, it would be unsound to conclude that $(@\tau_1) \leftarrow\!\!+ \tau_2$ is a subtype of τ_1 if τ_1 itself were exact; exact types must have no nontrivial subtypes.)

Rule 18 is the resulting subtyping rule for extended types. This rule is particularly

Well-formed Contexts

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

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

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

Well-Formed Types

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash \{\!|\,|\!\} :: \mathsf{In}} \tag{4}$$

$$\frac{\Gamma \vdash \mathrm{ok} \qquad \Gamma = \ldots, \alpha \preceq \tau :: \kappa, \ldots}{\Gamma \vdash \alpha :: \kappa} \tag{5}$$

$$\frac{\Gamma \vdash \tau :: \mathsf{In}}{\Gamma \vdash @\tau :: \mathsf{Ex}} \tag{6}$$

$$\frac{\Gamma \vdash \tau :: \mathsf{Ex} \qquad \Gamma \vdash \sigma :: \mathsf{Ty}}{\Gamma \vdash \tau \leftarrow\!\!+ \sigma :: \mathsf{In}} \tag{7}$$

$$\frac{\Gamma \vdash \tau :: \mathsf{Ty} \qquad \Gamma \vdash \sigma :: \mathsf{Ty}}{\Gamma \vdash \tau \Rightarrow \sigma :: \mathsf{Ty}} \tag{8}$$

$$\frac{\Gamma \vdash \tau_1 :: \kappa \qquad \Gamma, \alpha \preceq \tau_1 :: \kappa \vdash \tau_2 :: \mathsf{Ty}}{\Gamma \vdash (\exists \alpha \preceq \tau_1 :: \kappa . \tau_2) :: \mathsf{Ty}} \tag{9}$$

$$\frac{\Gamma \vdash \tau :: \kappa_1 \qquad \Gamma \vdash \kappa_1 \preceq \kappa_2}{\Gamma \vdash \tau :: \kappa_2} \tag{10}$$

Fig. 6. Static Semantics of COI

Subkinding

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash \mathsf{Ex} \preceq \mathsf{Ty}} \tag{11}$$

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash \mathsf{In} \preceq \mathsf{Ty}} \tag{12}$$

$$\frac{\Gamma \vdash \mathrm{ok}}{\Gamma \vdash \kappa \preceq \kappa} \tag{13}$$

Subtyping

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

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

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

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

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

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

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

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

Fig. 7. Static Semantics of COI (continued)

Well-Formed Expressions

$$\frac{\begin{array}{c}\Gamma, s{:}\{\!|\, \tau_1, \ldots, \tau_n \,|\!\} \vdash e_1 : \tau_1 \\ \vdots \\ \Gamma, s{:}\{\!|\, \tau_1, \ldots, \tau_n \,|\!\} \vdash e_n : \tau_n\end{array}}{\Gamma \vdash \mathsf{obj}\ s.\{\!|e_1{:}\tau_1, \ldots, e_n{:}\tau_n|\!\} : @\{\!|\, \tau_1, \ldots, \tau_n \,|\!\}} \tag{22}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_1 \Rightarrow \tau}{\Gamma \vdash e_1.e_2 : \tau} \tag{23}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash \tau_1 :: \mathsf{Ex} \\ \Gamma, s{:}\tau_1 \leftarrow\!\!+ \tau_2 \vdash e_2 : \tau_2\end{array}}{\Gamma \vdash e_1 \leftarrow\!\!+ (s) = e_2 : \tau_2 : @(\tau_1 \leftarrow\!\!+ \tau_2)} \tag{24}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_1 \Rightarrow \tau_3 \\ \Gamma \vdash \tau_1 \preceq \tau_1' :: \mathsf{Ty} \qquad \Gamma \vdash \tau_1' :: \mathsf{In} \\ \Gamma, s{:}\tau_1' \vdash e_3 : \tau_3\end{array}}{\Gamma \vdash e_1 \leftarrow e_2 (s) = e_3 : \tau_1} \tag{25}$$

$$\frac{\Gamma \vdash \tau_1 :: \mathsf{Ex} \qquad \Gamma \vdash \tau_2 :: \mathsf{Ty}}{\Gamma \vdash \mathsf{next\_index}(\tau_1) : (\tau_1 \leftarrow\!\!+ \tau_2) \Rightarrow \tau_2} \tag{26}$$

$$\frac{\Gamma \vdash \tau_1 :: \mathsf{Ty} \quad \cdots \quad \Gamma \vdash \tau_n :: \mathsf{Ty} \qquad k \in 1..n}{\Gamma \vdash \ulcorner k \urcorner : \{\!|\, \tau_1, \ldots, \tau_n \,|\!\} \Rightarrow \tau_k} \tag{27}$$

$$\frac{\Gamma \vdash \mathrm{ok} \qquad s \in \mathrm{dom}\, \Gamma}{\Gamma \vdash s : \Gamma(s)} \tag{28}$$

$$\frac{\Gamma \vdash \sigma \preceq \tau_1 :: \kappa \qquad \Gamma \vdash e : \tau_2[\sigma/\alpha] \qquad \Gamma \vdash (\exists \alpha \preceq \tau_1 :: \kappa.\tau_2) :: \mathsf{Ty}}{\Gamma \vdash (\mathsf{pack}\ \sigma\ \mathsf{and}\ e\ \mathsf{as}\ \exists \alpha \preceq \tau_1 :: \kappa.\tau_2) : (\exists \alpha \preceq \tau_1 :: \kappa.\tau_2)} \tag{29}$$

$$\frac{\Gamma \vdash e_1 : (\exists \alpha \preceq \tau_1 :: \kappa.\tau_2) \qquad \Gamma, \alpha \preceq \tau_1 :: \kappa, x{:}\tau_2 \vdash e_2 : \tau \qquad \Gamma \vdash \tau :: \mathsf{Ty}}{\Gamma \vdash \mathsf{open}\ e_1\ \mathsf{as}\ \alpha\ \mathsf{and}\ x\ \mathsf{in}\ e_2 : \tau} \tag{30}$$

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

Fig. 8. Static Semantics of COI (cont.)

important because the abbreviations from Section 2.1 yield the derived rule

$$\frac{\Gamma \vdash \{\!|\, \sigma_1, \ldots, \sigma_n \,|\!\} :: \mathsf{In} \qquad \Gamma \vdash \sigma_{n+1} :: \mathsf{Ty}}{\Gamma \vdash \{\!|\, \sigma_1, \ldots, \sigma_{n+1} \,|\!\} \preceq \{\!|\, \sigma_1, \ldots, \sigma_n \,|\!\} :: \mathsf{In}},$$

which is exactly (prefix) width subtyping for object types.

Rule 19 handles subtyping for method index 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. To see this, assume that $\sigma_1 \preceq \sigma_2$. First, suppose there was an object with a field of type σ_1, and that the index for this component had type $\tau \Rightarrow \sigma_1$. If index types were covariant in the component type, then this index would also have type $\tau \Rightarrow \sigma_2$ and hence one could override the field with a value of the supertype σ_2. This is unsound because other methods of the object may be expecting the field to contain a value of type σ_1 rather than of the more general type σ_2. Contravariance fails as well; if there was an object with a field of type σ_2 and its index of type $\tau \Rightarrow \sigma_2$ were treated as having type $\tau \Rightarrow \sigma_1$ then one could use the index to access the field expecting to obtain a value of the subtype σ_1. (This invariance corresponds to the usual lack of depth subtyping in calculi with method override [Abadi and Cardelli 1996].) Index types are contravariant in their domain type, because an index suitable for any object with components of certain types is equally suitable for an object with more components.

Existential types subtype covariantly as shown in Rule 20. Finally, Rule 21 is a subsumption rule for subtyping that parallels Rule 10 for type well-formedness.

2.3.3 *Well-Formed Expressions.* In Rule 22, which handles the typing of object values, each method is type checked assuming that the variable s will contain the object itself. The resulting type of this object value can be exact because all of the components are syntactically apparent. However, while type checking methods s is assumed to have an inexact object type, i.e., we assume only that the object contains *at least* the components e_1 through e_n. By the time the the method is eventually invoked the object may have been extended with still more components, so it would be unsound for methods to assume they know the exact type of the object in which they appear. Because only objects with exact types may be extended, it follows that methods may not directly extend the objects which contain them.

Although Rule 23 for typing method selection does not explicitly mention object types, the fact that there is an index value of the right type guarantees that e_1 is an object.

Rule 24 type checks extensions for objects of exact types. As in Rule 22, when the code for the new method is type checked the type of the variable representing the containing object is not assumed to be known exactly — even though the type of the object resulting from this extension can be determined exactly. Extension is still not quite type-preserving, because it does not preserve exact types. Extension does preserve all (inexact) supertypes of the object's exact type, however, and this is enough to obtain useful variants of the class type and mixin encodings discussed in Sections 1.1 and 1.2.

Rule 25 for method override is slightly more complex than one might expect. Override is a type-preserving (and even principal-type-preserving) operation, and

override is sound for any object whether or not its exact type is known. However, when type checking the code e_3 for the new method body, again the system cannot assume that s has the same type as the object being extended if that type is exact. To allow override to preserve all types, including exact types, the object's type is merely required have an inexact supertype τ_1'. This is used as the type of s when type checking the method body because the object will still have type τ_1' when the method is invoked, regardless of intervening overrides and extensions.

Rule 26 types the expression $\mathsf{next\_index}(\tau_1)$, which yields the index at which a new method would be found if it were added to an object of the exact type τ_1. If this new method returns a value of type τ_2, then the extended object would have type $\tau_1 \leftarrow\!\!+ \tau_2$; hence, the index returned by the next_index operator is usable with objects having type $\tau_1 \leftarrow\!\!+ \tau_2$ and, when used, accesses a component of type τ_2.

Using these rules, one can show that every line in the example evaluation at the end of Section 2.2 has type `int`. Further, the object being selected from (both in the first line and thereafter) can be shown to have type $\{\!|\, \texttt{int}, \texttt{int} \,|\!\}$, i.e., $(@((@\{\!|\,|\!\}) \leftarrow\!\!+ \texttt{int})) \leftarrow\!\!+ \texttt{int}$.

2.4 Soundness

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

THEOREM 1. *(Type Preservation) If* $\vdash e : \tau$ *and* $e \rightsquigarrow e'$ *then* $\vdash e' : \tau$.

THEOREM 2. *(Progress) If* $\vdash e : \tau$ *then either* e *is a value or there exists* e' *such that* $e \rightsquigarrow e'$.

Proofs can be found in the appendix.

3. APPLYING COI

Figure 9 shows a revised encoding for the inheritance example with class types, now expressed with COI primitives. As in Figure 2 the `POINT` interface specifies a partially abstract implementation type `T`, but instead of requiring `T` to be a subtype of a labeled object type, we merely require that `T` be inexact, and that there be an index value named `get1` that selects an integer method in values of type `T`. The type for the constructor function `new` then specifies not only that values returned satisfy type `T`, but guarantees they have exactly the type `T`.

The `Point` module now provides an unlabeled object type `T` that is the implementation type for this class's objects. The module contains two more values than before, the index values for the two components.[4] The code for `new` is as before, except that no labels are specified for the two components. Finally, although the code for `eq` looks unchanged from Figure 2, the method invocations `o1.x` and `o2.x` are no longer choosing the method with label `x` in the `o1` and `o2` objects; they are invoking the method whose index is given by the variable `x` defined five lines earlier.

[4] `T` is completely known within `Point` so the two indices could have been defined as $\ulcorner 1 \urcorner$ and $\ulcorner 2 \urcorner$ respectively, instead of using next_index. Because `Point.T` is externally abstract, the type system would still require the use of next_index to obtain index values in the implementation of the subclass `Point2D`.

```
signature POINT = sig
                    type T ⪯ {| |} :: In
                    val get1 : T => int
                    val new  : int -> @T
                    val eq   : T*T -> bool
                  end =

structure Point :> POINT =
     struct
       type T = {| int, int |}
       val x    = next_index(@{| |})
       val get1 = next_index(@{|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


signature POINT2D = sig
                      type T ⪯ Point.T :: In
                      val get1 : T => int
                      val get2 : T => real
                      val new  : int*real -> @T
                    end =

structure Point2D :> POINT2D =
     struct
       type T = @Point.T ←+ (real,real)
       val x    = next_index(@Point.T)
       val get2 = next_index(@((@Point.T)←+real))
       val get1 = Point.get1
       fun new (x1, x2) = Point.new(x1)
                            ←+(s) = x2  : real
                            ←+(s) = s.x : real
     end
```

Fig. 9. Class Encodings using COI Objects

In contrast to the Figure 2 encoding, a strict subtype `Point2D.T` of the abstract type `Point.T` can be defined, and this relationship is revealed in the `POINT2D` interface while still keeping `Point2D.T` abstract. The subclass `Point2D` extends `Point.T` with two new components, whose indices are `x` and `get2` respectively. The `POINT2D` interface exposes the index values `get1` and `get2`, but not the index for either of the `x` fields, keeping them private.

Given these definitions one can conclude that `Point.new(0)` has type `@Point.T` and hence also type `Point.T`. The function call `Point2D.new(4, 7.5)` has the types `@Point2D.T` and `Point2D.T` and `Point.T`, but does not satisfy the type `@Point.T` as it does not have exactly the same structure as an object created by the `Point` class. Given an object `p2 : Point2D.T`, one can then invoke `p2.(Point2D.get1)` to obtain an integer or `p2.(Point2D.get2)` to obtain a real. By subsumption and the contravariance of index types one could also invoke `p2.(Point.get1)` to get

```
(* Setup: an SML type for colors *)
datatype color = Red | Blue

(* Mixin definition *)
functor ColorMix(P : sig                          (* Argument interface *)
                        type T :: In
                        val new : int -> @T
                      end)
          :> sig                                  (* Result interface *)
               type T ≼ P.T :: In
               val getc : T => color
               val new  : int*color -> @T
             end =

   struct
      type T   = @P.T ←+ (color, color)          (* Adding 2 color components *)
      val c    = next_index(@P.T)                (* Index of c field *)
      val getc = next_index(@((@P.T)←+color))    (* Index of accessor *)
      fun new(z0:int, c0:color) =                (* Constructor *)
          P.new(z0) ←+ c(s) = c0 : color
                    ←+ getc(s) = s.c : color
   end


(* Sample code: building and using a subclass: *)
signature CPOINT = sig
                     type T ≼ Point.T :: In
                     val get1 : T => int
                     val getc : T => color
                     val new  : int*color -> @T
                   end

structure ColorPoint :> CPOINT =
   let
      structure NewStuff = ColorMix(Point)       (* Apply the mixin to Point *)
   in
      struct
         type T    = NewStuff.T     (* Copy everything in the mixin's result *)
         val  getc = NewStuff.getc  (* into the ColorPoint module            *)
         val  new  = NewStuff.new

         val get1 = Point.get1      (* And then copy anything else inherited *)
                                    (* without change, in this case just the *)
                                    (* offset for the get1 accessor method   *)
      end (* struct *)
   end (* let *)
```

Fig. 10. A Mixin which Adds Color

that same integer; It would have been possible to rename or omit this index from the subclass interface without breaking the subtyping relationship between `Point.T` and `Point2D.T`.

Next, if classes are SML modules, then mixins or parameterized classes can be

encoded as SML functors, which are simply module-level functions. Figure 10 shows a mixin for adding a private color field along with an accessor method. The `ColorMix` functor expects its argument `P` to be a the encoding of a class, i.e., a module containing an inexact type `T` and a constructor function creating objects of type `T`.[5] The functor's result provides a subtype of the given type `P.T`, a constructor function for creating values of this subtype, and an index for at a new color component in values of the subtype.

The implementation of the `ColorMix` functor extends the given class with two components: a color field, and an accessor method for that field.

Figure 10 also includes code that applies the `ColorMix` functor to the `Point` class of Figure 9 to obtain a subclass `ColorPoint` of colored one-dimensional points. Specifically, the definition of `ColorPoint` creates a temporary structure `NewStuff` by applying the mixin to the `Point` module to obtain `T`, `getc`, and `new`, and then copies this into the `ColorPoint` module along with the inherited index `Point.get1`. (This extra copying is necessary because SML functors must return a module with a fixed set of module components; they cannot generically extend their argument structure.)

Thus, one can invoke `ColorPoint.new(3, Red)` to obtain a value satisfying the types `@ColorPoint.T` and `ColorPoint.T` and `Point.T`, and on such an object one can invoke methods with indices `ColorPoint.get1` and `ColorPoint.getc`. It is also possible to compare two such values as one-dimensional points using the function `Point.eq`.

4. VARIANCE ANNOTATIONS

As mentioned in Section 2.3.2, the type $\tau_1 \Rightarrow \tau_2$ must be invariant in its range type τ_2 because it provides both the capability to override and to invoke methods. However, Abadi and Cardelli [1996] note that by using *variance annotations* to restrict override or invocation on a method-by-method basis, one can obtain more subtypings. This suggests that COI could be extended by annotating index types with a variance a, where $a \in \{+, 0, -\}$. An index of type $\tau_1 \Rightarrow^{+} \tau_2$ could be used to invoke but not override a method and so such types can be covariant in τ_2. An index of type $\tau_1 \Rightarrow^{-} \tau_2$ could be used to override but not invoke a method, and such types could be contravariant in τ_2. Finally, indices of type $\tau_1 \Rightarrow^{0} \tau_2$ could be used to either override or invoke a method and thus remain invariant, as in the system originally presented.

The annotation must be part of the index type rather than kept within the object type because at the time when access or override is type checked the object type may not be statically known (as in code that uses values of the abstract type `Point.T` from Figure 9).

The necessary changes for adding variances are shown in Figure 11. The dynamic semantics is unaffected and the language remains sound.

[5] Formally the code expects the argument class's constructor function to take a single integer argument. The encoding could be generalized to expect an arbitrary type `initial` and a constructor function `new : initial -> @T`. Then the module returned would include a constructor function `new : initial*color -> @T`

Changes to Syntax

$$\begin{array}{rcl} \tau, \sigma ::= & \ldots & \\ & \mid & \tau \Rrightarrow^a \tau \qquad a \in \{+, -, 0\} \end{array}$$

Changes to Well-Formed Types

$$\frac{\Gamma \vdash \tau :: \mathsf{Ty} \qquad \Gamma \vdash \sigma :: \mathsf{Ty} \qquad a \in \{+, -, 0\}}{\Gamma \vdash \tau \Rrightarrow^a \sigma :: \mathsf{Ty}} \tag{8}$$

Changes to Subtyping

$$\frac{\Gamma \vdash \tau_1' \preceq \tau_1 :: \mathsf{Ty} \qquad \Gamma \vdash \tau_2 :: \mathsf{Ty}}{\Gamma \vdash (\tau_1 \Rrightarrow^0 \tau_2) \preceq (\tau_1' \Rrightarrow^0 \tau_2) :: \mathsf{Ty}} \tag{19}$$

$$\frac{\Gamma \vdash \tau_1' \preceq \tau_1 :: \mathsf{Ty} \qquad \Gamma \vdash \tau_2 \preceq \tau_2' :: \mathsf{Ty} \qquad a \in \{+, 0\}}{\Gamma \vdash (\tau_1 \Rrightarrow^a \tau_2) \preceq (\tau_1' \Rrightarrow^+ \tau_2') :: \mathsf{Ty}} \tag{19'}$$

$$\frac{\Gamma \vdash \tau_1' \preceq \tau_1 :: \mathsf{Ty} \qquad \Gamma \vdash \tau_2' \preceq \tau_2 :: \mathsf{Ty} \qquad a \in \{-, 0\}}{\Gamma \vdash (\tau_1 \Rrightarrow^a \tau_2) \preceq (\tau_1' \Rrightarrow^- \tau_2') :: \mathsf{Ty}} \tag{19"}$$

Changes to Well-Formed Expressions

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_1 \Rrightarrow^+ \tau}{\Gamma \vdash e_1.e_2 : \tau} \tag{23}$$

$$\frac{\begin{array}{c} \Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_1 \Rrightarrow^- \tau_3 \\ \Gamma \vdash \tau_1 \preceq \tau_1' :: \mathsf{Ty} \qquad \Gamma \vdash \tau_1' :: \mathsf{In} \\ \Gamma, s{:}\tau_1' \vdash e_3 : \tau_3 \end{array}}{\Gamma \vdash e_1 \leftarrow e_2(s) = e_3 : \tau_1} \tag{25}$$

$$\frac{\Gamma \vdash \tau_1 :: \mathsf{Ex}}{\Gamma \vdash \mathsf{next\_index}(\tau_1) : (\tau_1 \leftarrow\!\!+ \tau_2) \Rrightarrow^0 \tau_2} \tag{26}$$

$$\frac{\Gamma \vdash \tau_1 :: \mathsf{Ty} \quad \cdots \quad \Gamma \vdash \tau_n :: \mathsf{Ty} \qquad k \in 1..n}{\Gamma \vdash \ulcorner k \urcorner : \{\!| \tau_1, \ldots, \tau_n |\!\} \Rrightarrow^0 \tau_k} \tag{27}$$

Fig. 11. Changes for Variance Annotations

5. ENCODING FIRST-ORDER RS OBJECTS

The first-order RS system (i.e., without a notion of `SelfType` or `ThisType`) can be encoded within COI as well. COI objects were obtained by stripping the dictionaries from RS objects. First-order RS objects with labeled components can be recovered by packaging a bare COI object with an appropriate dictionary, which can be represented as an ordinary labeled record of index values. The connection between the object and the index values can be enforced by using existentials. Then, for

example, the object type

$$\{\!|\ \texttt{get1 : int}\ |\!\}$$

could be encoded as the type

$$\exists\alpha\preceq\{\!|\,|\!\}::\mathsf{In}.\ (@\alpha \times \{\texttt{get1} : \alpha\Rightarrow\texttt{int}\}).$$

Such a type classifies values having both a type α and a pair containing a (dictionary-less) COI object having some unknown implementation type $@\alpha$, and a record of indices suitable for this object; the record represents an RS dictionary. In this case, the dictionary is required to have one component, giving the index of a method `get1`.

Similarly, the object type

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

could be encoded as the type

$$\exists\alpha\preceq\{\!|\,|\!\}::\mathsf{In}.\ (@\alpha \times \{\texttt{get1} : \alpha\Rightarrow\texttt{int}, \texttt{get2} : \alpha\Rightarrow\texttt{real}\}).$$

Assuming width subtyping for records, the translation of two RS subtypes yields COI subtypes.

Objects can be translated correspondingly; the RS object

$$\mathsf{obj}\ s.\{\!|\texttt{31 : int, get1 = }s\texttt{.1 : int}|\!\}_{[\mathtt{x}\mapsto 1,\mathtt{get1}\mapsto 2]}$$

would become an existential package containing the underlying implementation type for the object, $\{\!|\,\texttt{int}, \texttt{int}\,|\!\}$, along with a pair containing the corresponding unlabeled object and a record of the two indices:

$$\begin{array}{l}\mathsf{pack}\ \{\!|\,\texttt{int}, \texttt{int}\,|\!\}\ \mathsf{and}\ (\mathsf{obj}\ s.\{\!|31 : \texttt{int},\ s.\ulcorner 1\urcorner : \texttt{int}|\!\}, \{\mathtt{x} = \ulcorner 1\urcorner, \mathtt{get1} = \ulcorner 2\urcorner\})\ \mathsf{as}\\ \qquad \exists\alpha\preceq\{\!|\,|\!\}::\mathsf{In}.\ (\alpha \times \{\mathtt{x} : \alpha\Rightarrow\mathtt{int}, \mathtt{get1} : \alpha\Rightarrow\mathtt{int}\}).\end{array}$$

The encoding of RS by-label method invocation must open the existential, project the index value with that label from the dictionary record, and invoke that method of the underlying COI object. Encoding of method override proceeds similarly, except that the overridden COI object must then be repackaged with the original dictionary. The encoding of object extension is then similar to method override, except that the extended COI object must be packaged with a dictionary record with a new index added.

The extensible-object calculus of Fisher and Mitchell [1995] has two sorts of labeled object types: *pro* types, which permit override and extension but no width or depth subtyping, and *obj* which permit both forms of subtyping but neither override nor extension. Further, every *pro* type is a subtype of the corresponding *obj* type. The above encoding for labeled objects can be viewed as a sort of *pro* type that supports width subtyping. These COI types also have supertypes which forbid extension and override but allow width and depth subtyping. Corresponding to the type

$$\exists\alpha\preceq\{\!|\,|\!\}::\mathsf{In}.\ (@\alpha \times \{\texttt{get1} : \alpha\Rightarrow\texttt{int}\}),$$

used as the encoding for an extensible object type, would be the supertype

$$\exists\alpha\preceq\{\!|\,|\!\}::\mathsf{In}.\ (\alpha \times \{\texttt{get1} : \alpha \Rightarrow^{+} \texttt{int}\})$$

that both forbids extension (since the underlying object no longer can be given an exact type) and forbids override (since the index type has a variance annotation). Encodings of in-between types (i.e., that permit extension but not override) are possible as well.

These existential encodings are very reminiscent of the object encodings of Pierce and Turner [1993], but there the value packaged with a record was only the state (fields) of the object, while the method code resided in an attached record of functions. In the above encodings, all the code resides in the object value, and the record contents are merely capabilities granting access to the fields and methods within that object.

Going back to the class encodings, we see that the objects created by the `Point` class encoding in Figure 2 can be treated either as having the abstract type `Point.T` or the less-precise interface type $\{|\,$`get1 : int`$\,|\}$. The objects created by the revised encoding of `Point.new` in Figure 9 can be treated either as having the exact type @`Point.T` (i.e., created by the `Point` class) or type `Point.T` (i.e., created by `Point` or a derived class), but there is no corresponding interface type. This could be addressed in COI by extending the `Point` structure with a run-time coercion function which transforms objects of the abstract type `Point.T` into objects with interface types by adding dictionaries and existential types as sketched above.

6. CLOSING REMARKS

6.1 Global Uniqueness

Why not simply require some sort of global-uniqueness property for labels added to objects, thus trivially avoiding unintended clashes?

If the uses of object extension are sufficiently restricted then it may be possible to statically check that no object is extended with a given label more than once. A special case of this would be a conventional class-based language, where each component name logically consists both of the programmer-specified name and the name of the class in which it was added. As long as classes have globally unique names — usually checkable at link-time — then there can be no collisions. A `Point` class and its subclass `Point2D` can both have `x` fields because the former is identified internally by the interpreter or compiler as `Point::x` and the latter `Point2D::x`.

There is no doubt this is practical, but it is theoretically somewhat inelegant. Soundness depends on global invariants enforced by restricting classes to be relatively static entities; for example, it depends on there being no way to get object extension (inheritance) inside a run-time loop. This may not be a problem if classes are modeled as top-level primitives, but trying to maintain this sort of reasoning formally when lower-level, more flexible constructs are available (as in RS or COI) is significantly more difficult.

6.2 Implementation Considerations

The class encodings of Section 3 appear to make method invocation slightly less efficient than implementations found in C++ and other traditional class-based languages, because the offsets of fields and methods within objects are run-time values rather than constants.

An important goal of the work presented here is to provide a semantic basis for compilation of subclasses with minimal knowledge of superclasses. The advantage is that superclasses can be extended or refactored without requiring recompilation of subclasses (addressing the so-called fragile base class problem). The cost of this flexibility is the possibility that index values might not be determined until link-time, or possibly even run time (e.g., if the construction of classes or the application of mixins is allowed to depend on run-time control flow)

If one is willing to give up complete separate compilation and use incremental or full-program compilation techniques, then standard cross-module optimization techniques are likely to statically determine all offsets. In this case, the offsets can again be embedded in code as constants. This appears to be the approach used to implement Binary Compatibility guarantees of Java [Gosling et al. 2000].

Because the dynamic semantics as presented for $\mathsf{next\_index}(\tau)$ depends on knowing the type σ, either the compiler must be able to determine τ at compile-time or else some object types must be generated and tested at run time [Harper and Morrisett 1995; Morrisett 1995].

To avoid run-time type passing and analysis by the next_index operator, one could instead use run-time term values representing the layout — or at least the size — of objects. This could be formalized using the representation type framework of Crary et al. [1998]. In a simpler form, the system could be extended with a new type $\mathsf{SizeOf}(\tau)$ classifying object-size values: a constant $\mathsf{zero} : \mathsf{SizeOf}(@\{\!|\,|\!\})$ that holds the object-size of the empty object, and an operator

$$\mathsf{inc} : \forall\alpha{::}\mathsf{Ex}.\forall\beta{::}\mathsf{Ty}.(\mathsf{SizeOf}(\alpha) \to \mathsf{SizeOf}(\alpha{\leftarrow\!\!+}\beta))$$

for incrementing object-size values. Then the next_index operator can be modified to have type

$$\forall\alpha{::}\mathsf{Ex}.\forall\beta{::}\mathsf{Ty}.(\mathsf{SizeOf}(\alpha) \to ((\alpha{\leftarrow\!\!+}\beta){\Rightarrow}\beta))$$

and can, at run time, depend only on object-size *values* for objects — rather than the types themselves — to determine the appropriate index value. Each class module as encoded in Section 3 would then need not just a type `T` and a constructor function, but also a value `T_size` of type $\mathsf{SizeOf}(\mathtt{T})$.

6.3 Other Related Work

The COI calculus has connections with a number of ideas which have appeared in other sources, in addition to those systems already mentioned.

COI is somewhat reminiscent of the untyped language λink_{ς} studied by of Fisher et al. [2000]. 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.

The two different extensions of the RS system to allow covariant self-types both used dictionaries as first-class values [Riecke and Stone 1998; 2002]. Later, Vouillon [2001] significantly extended the original 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. It might also be possible to

extend the ideas used to encode RS into COI to encode these other more general calculi.

The distinction between exact and inexact object types in COI is somewhat parallel to the distinction between (extensible) *pro* and (inextensible) *obj* types of Fisher and Mitchell [1995]. For example, for soundness both calculi must type check 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 of inexact types, and it would not be intrinsically unsound to permit extension for objects of inexact types.

Other approaches have been suggested to achieve by-name subtyping of classes using object calculi. Fisher and Mitchell [1998] first define all the classes with exposed interfaces, and then use after-the-fact abstraction 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. [1999] get the proper subtyping behavior by encoding the class hierarchy within the types of objects; this requires every class know the identity of all superclasses and requires specific object layouts containing extra indirections. Alternatively, Bono et al. [1999] and Fisher and Reppy [2000] have studied the difficulties in encoding classes with standard extensible objects and have suggested extending object calculi with primitive class mechanisms.

Ohori [1995] describes a typed intermediate language for compiling polymorphic record calculi which uses integer indices corresponding to record components as values. The goals and mechanisms seem very different, however, as the connections between indices and records are enforced primarily in the kind system, rather than in the type system as in COI. Further, languages such as Ohori's that use row-polymorphism generally do not support subsumption.

The C++ language [Stroustrup 1997] includes pointers-to-member, which are essentially equivalent to the indices here.

Bruce [1997] has applied exact types in object-oriented languages, primarily in the context of typing binary methods.

Finally, the idea of avoiding name clashes by avoiding labels in objects also occurs in multimethod approaches to object-oriented programming [DeMichiel and Gabriel 1987; Chambers 1992]. 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. However, the arguments of these multimethods remain collections of values (fields), and some similar mechanism may be necessary distinguish these components at run-time.

6.4 Future Work

There are a number of ways in which this work could be extended. Many systems of object calculi have been extended to allow self-types, at least in covariant positions, but there has been no work as yet on adding covariant self-types to COI. At a minimum, such a system would seem to require index values to have polymorphic types, e.g., the index of a `clone` method in an object of type `t` would require a type along the lines of $\forall\alpha\preceq\mathtt{t}.\alpha\Rightarrow\alpha$.

Alternatively, one could go beyond object extension to object concatenation, in which two objects are merged into a single object. Multiple inheritance in C++

can be viewed as a concatenation of objects, and it appears that the COI system already contains most of the mechanisms necessary to directly model the classical approach to implementing C++ multiple inheritance. (Extra bookkeeping of offsets within an object [Lippman 1996] is used; these offsets could likely be represented by index values.)

It would be useful to know to how far the encodings of Section 3 can handle features of conventional class-based languages. Adding a method to invoke overridden methods (`super`) seems likely to be possible. Downcasting and run-time class tests could be simulated with the addition of a run-time tagging mechanism, e.g., along the lines of the extensible datatype `exn` in Standard ML [Milner et al. 1997] or as extended to subtyping by Reppy and Riecke [1996].

6.5 Conclusion

By removing all labels from objects, the COI calculus retains the benefits of the RS system — specifically the ability to add new methods without needing to know of all pre-existing methods — but further permits encodings of useful class and mixin constructs with by-name subtyping. 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, or even implementations of object calculi such as RS.

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

ACKNOWLEDGMENTS

This work has benefited significantly from the helpful comments of several anonymous reviewers.

REFERENCES

Abadi, M. 1994. Baby Modula-3 and a theory of objects. *Journal of Functional Programming 4,* 2, 249–283.

Abadi, M. and Cardelli, L. 1996. *A Theory of Objects*. Springer-Verlag.

Bono, V., Patel, A. J., Shmatikov, V., and Mitchell, J. C. 1999. A core calculus of classes and objects. In *Fifteenth Confertence on the Mathematical Foundations of Programming Semantics*.

Bracha, G. and Cook, W. 1990. Mixin-based inheritance. In *Proceedings of Conference on Object-Oriented Programming: Systems, Languages, and Applications*. 303–311.

Bruce, K. B. 1997. Increasing Java's expressiveness with ThisType and match-bounded polymorphism. Tech. rep., Williams College.

Cardelli, L. and Mitchell, J. C. 1989. Operations on records. In *Proceedings of the Fifth Conference on the Mathematical Foundations of Programming Semantics*. Springer Verlag.

Chambers, C. 1992. Object-oriented multi-methods in Cecil. In *Proceedings of ECOOP '92 European Conference on Object-Oriented Programming*. LNCS, vol. 615. 33–56.

Crary, K., Weirich, S., and Morrisett, G. 1998. Intensional polymorphism in type-erasure semantics. In *1998 International Conference on Functional Programming*. 301–312.

Crary, K. F. 1998. Type-theoretic methodology for practical programming languages. Ph.D. thesis, Department of Computer Science, Cornell University.

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

Di Gianantonio, P., Honsell, F., and Liquori, L. 1998. A lambda calculus of objects with self-inflicted extension. In *Proceedings of the 1998 ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '98)*. 166–178.

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

Fisher, K. and Mitchell, J. C. 1995. A delegation-based object calculus with subtyping. In *Fundamentals of Computation Theory (FCT'95)*. Number 965 in Lecture Notes in Computer Science. Springer-Verlag, 42–61.

Fisher, K. and Mitchell, J. C. 1998. On the relationship between classes, objects, and data abstraction. *Theory and Practice of Object Systems 4,* 1 (January), 3–25.

Fisher, K. and Reppy, J. 2000. Extending Moby with inheritance-based subtyping. In *14th European Conference on Object-Oriented Programming*. Vol. 1850. 83–107.

Fisher, K., Reppy, J. H., and Riecke, J. G. 2000. A calculus for compiling and linking classes. In *European Symposium on Programming*. 135–149.

Flatt, M., Krishnamurthi, S., and Felleisen, M. 1998. Classes and mixins. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages (POPL '98)*. 171–183.

Gosling, J., Joy, B., Steele, G., and Bracha, G. 2000. *The Java Language Specification, Second Edition*. Addison-Wesley.

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

League, C., Shao, Z., and Trifonov, V. 1999. Representing Java classes in a typed intermediate language. In *International Conference on Functional Programming*. 183–196.

Lippman, S. B. 1996. *Inside the C++ Object Model*. Addison-Wesley.

Liquori, L. 1997. An extended theory of primitive objects: First order system. In *Proceedings of ECOOP-97, International European Conference on Object Oriented Programming*, M. Aksit and S. Matsuoka, Eds. Number 1241 in Lecture Notes in Computer Science. Springer-Verlag.

Milner, R., Tofte, M., Harper, R., and MacQueen, D. 1997. *The Definition of Standard ML (Revised)*. MIT Press.

Morrisett, G. 1995. Compiling with Types. Ph.D. thesis, Carnegie Mellon University School of Computer Science. Available as CMU Technical Report CMU-CS-95-226.

Nelson, G., Ed. 1991. *Systems Programming with Modula-3*. Prentice Hall.

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

Pierce, B. C. 2002. *Types and Programming Languages*. MIT Press.

Pierce, B. C. and Turner, D. 1993. Object-oriented programming without recursive types. In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages (POPL '93)*. 299–312.

Reppy, J. and Riecke, J. 1996. Simple objects for Standard ML. In *Proceedings of the ACM 1996 Conference on Programming Language Design and Implementation (PLDI '96)*. 171–180.

Riecke, J. C. and Stone, C. A. 1998. Privacy via subsumption. In *Fifth International Workshop on Foundations of Object-Oriented Programming (FOOL 5)*.

Riecke, J. G. and Stone, C. A. 2002. Privacy via subsumption. *Information and Computation 172*, 2–28.

Stroustrup, B. 1997. *The C++ Programming Language*. Addison-Westley.

Vouillon, J. 2001. Combining subsumption and binary methods: An object calculus with views. In *Proceedings of the 28th ACM Symposium on Principles of Programming Languages (POPL '01)*. 290–302.

Wright, A. and Felleisen, M. 1991. A syntactic approach to type soundness. Tech. Rep. TR91-160, Department of Computer Science, Rice University.

Proof of the Soundness Theorems

LEMMA 3. *(Validity)*

(1) *If* $\Gamma \vdash \kappa_1 \preceq \kappa_2$ *then* $\Gamma \vdash ok$.
(2) *If* $\Gamma \vdash \tau :: \kappa$ *then* $\Gamma \vdash ok$.
(3) *If* $\Gamma \vdash \tau_1 \preceq \tau_2 :: \kappa$ *then* $\Gamma \vdash \tau_1 :: \kappa$, $\Gamma \vdash \tau_2 :: \kappa$, *and* $\Gamma \vdash ok$.
(4) *If* $\Gamma \vdash e : \tau$ *then* $\Gamma \vdash \tau :: \mathsf{Ty}$ *and* $\Gamma \vdash ok$.

PROOF. By simultaneous induction on the assumed derivations. □

LEMMA 4. *(Weakening) Let* $\mathcal{J}$ *represent an arbitrary judgment form.*

(1) *If* $\Gamma \vdash \mathcal{J}$ *and* $\Gamma' \supseteq \Gamma$ *and* $\Gamma' \vdash ok$ *then* $\Gamma' \vdash \mathcal{J}$.
(2) *If* $\Gamma_1, s{:}\sigma, \Gamma_2 \vdash \mathcal{J}$ *and* $\Gamma_1 \vdash \tau \preceq \sigma :: \mathsf{Ty}$ *then* $\Gamma_1, s{:}\tau, \Gamma_2 \vdash \mathcal{J}$.

PROOF. Each part follows by induction on the proof of the first assumption. □

LEMMA 5. *(Substitution) Let* $\mathcal{J}$ *represent an arbitrary judgment form.*

(1) *If* $\Gamma_1, s{:}\tau, \Gamma_2 \vdash \mathcal{J}$ *and* $\Gamma \vdash v : \tau$ *then* $\Gamma_1, \Gamma_2 \vdash \mathcal{J}\big[v/s\big]$.
(2) *If* $\Gamma_1, \alpha{\preceq}\tau{::}\kappa, \Gamma_2 \vdash \mathcal{J}$ *and* $\Gamma \vdash \sigma \preceq \tau :: \kappa$ *then* $\Gamma_1, (\Gamma_2\big[\sigma/\alpha\big]) \vdash \mathcal{J}\big[\sigma/\alpha\big]$.

PROOF. Each part follows by induction on the proof of the first assumption. □

LEMMA 6. *(Inversion of Kinding)*

(1) *If* $\vdash \tau :: \mathsf{In}$ *then* $\tau = \{\!|\,|\!\}$ *or* $\tau = \tau_1 \leftarrow\!\!+ \tau_2$ *where* $\vdash \tau_1 :: \mathsf{Ex}$ *and* $\vdash \tau_2 :: \mathsf{Ty}$.
(2) *If* $\vdash \tau :: \mathsf{Ex}$ *then* $\tau = @\tau_1$ *where* $\vdash \tau_1 :: \mathsf{In}$.
(3) *If* $\vdash \tau :: \mathsf{In}$ *then* $\tau = \{\!|\, \sigma_1, \ldots, \sigma_n \,|\!\}$ *where* $n \geq 0$ *and* $\vdash \sigma_i :: \mathsf{Ty}$ *for all* $i \in 1..n$.
(4) *If* $\vdash \tau :: \mathsf{Ex}$ *then* $\tau = @\{\!|\, \sigma_1, \ldots, \sigma_n \,|\!\}$ *where* $n \geq 0$ *and* $\vdash \sigma_i :: \mathsf{Ty}$ *for all* $i \in 1..n$.
(5) *If* $\Gamma \vdash \tau_1 \Rightarrow \tau_2 :: \kappa$ *then* $\kappa = \mathsf{Ty}$.
(6) *If* $\Gamma \vdash \tau_1 \leftarrow\!\!+ \tau_2 :: \kappa$ *then* $\kappa = \mathsf{In}$ *or* $\kappa = \mathsf{Ty}$.
(7) *If* $\Gamma \vdash @\tau :: \kappa$ *then* $\kappa = \mathsf{Ex}$ *or* $\kappa = \mathsf{Ty}$.
(8) *If* $\Gamma \vdash (\exists \alpha{\preceq}\tau_1{::}\kappa_1.\tau_2) :: \kappa$ *then* $\kappa = \mathsf{Ty}$.

PROOF. Parts 1 and 2 follow by inspection of the type-validity rules. Parts 3 and 4 follow by simultaneous induction on τ. Parts 4–8 follow by induction on the proofs of the assumptions, and inspection of the subkinding rules. □

LEMMA 7. *(Subtypes)*

(1) *If* $\vdash \tau \preceq @\sigma :: \kappa$ *then* $\tau = @\sigma$.
(2) *If* $\vdash \tau \preceq \{\!|\, \sigma_1, \ldots, \sigma_n \,|\!\} :: \kappa$ *then* $\tau = \{\!|\, \sigma_1, \ldots, \sigma_n, \sigma_{n+1}, \ldots, \sigma_{n+m} \,|\!\}$ *or* $\tau = @\{\!|\, \sigma_1, \ldots, \sigma_n, \sigma_{n+1}, \ldots, \sigma_{n+m} \,|\!\}$ *for some* $m \geq 0$.
(3) *If* $\vdash \tau \preceq \sigma_1 \Rightarrow \sigma_2 :: \mathsf{Ty}$ *then* $\tau = \tau_1 \Rightarrow \sigma_2$ *with* $\vdash \sigma_1 \preceq \tau_1 :: \mathsf{Ty}$.
(4) *If* $\vdash \tau \preceq (\exists \alpha{\preceq}\sigma_1{::}\kappa_1.\sigma_2) :: \mathsf{Ty}$ *then* $\tau = (\exists \alpha{\preceq}\tau_1{::}\kappa_1.\tau_2)$ *with* $\vdash \sigma_1 \preceq \tau_1 :: \kappa_1$ *and* $\alpha{\preceq}\sigma_1{::}\kappa_1 \vdash \tau_2 \preceq \sigma_2 :: \mathsf{Ty}$.

PROOF. Each part can be proved by induction on the proof of the assumption, and cases on the last rule used.

(1) If the proof ends with a use of Rule 15 then the inductive hypothesis is used twice. By Lemma 6 the proof cannot end with a use of Rule 18.
(2) If the proof ends with a use of Rule 15 then we can apply the inductive hypothesis, and then either use part 1 or the inductive hypothesis again to get the desired result. The case for Rule 18 follows by the definition of unlabeled object types.
(3) If the proof ends with a use of Rule 15 then we can apply the inductive hypothesis twice and apply transitivity to obtain the desired results. By Lemma 6, the proof cannot end with a use of Rule 17 or 18.
(4) If the proof ends with a use of Rule 15 then we can apply the inductive hypothesis twice and apply part 2 of Lemma 4 and transitivity to obtain the desired results. By Lemma 6, the proof cannot end with a use of Rule 17 or 18.

The other possible cases are similar or trivial. □

LEMMA 8. *(Supertypes)*

(1) *If* $\Gamma \vdash @\tau \preceq \sigma :: \kappa$ *then* $\sigma = @\tau$ *or* $\Gamma \vdash \tau \preceq \sigma :: \kappa$.
(2) *If* $\Gamma \vdash (@\tau_1) \leftarrow\!\!+ \tau_2 \preceq \sigma :: \kappa$ *then* $\sigma = (@\tau_1) \leftarrow\!\!+ \tau_2$ *or* $\Gamma \vdash \tau_1 \preceq \sigma :: \kappa$.
(3) *If* $\Gamma \vdash \tau_1 \Rightarrow \tau_2 \preceq \sigma :: \kappa$ *then* $\sigma = \sigma_1 \Rightarrow \tau_2$ *and* $\Gamma \vdash \sigma_1 \preceq \tau_1 :: \mathsf{Ty}$.
(4) *If* $\Gamma \vdash (\exists \alpha \preceq \tau_1 :: \kappa_1 . \tau_2) \preceq \sigma :: \kappa$ *then* $\sigma = \exists \alpha \preceq \sigma_1 :: \kappa_1 . \sigma_2$ *where* $\Gamma \vdash \sigma_1 \preceq \tau_1 :: \kappa_1$ *and* $\Gamma, \alpha \preceq \sigma_1 :: \kappa_1 \vdash \tau_2 \preceq \sigma_2 :: \mathsf{Ty}$.
(5) *If* $\Gamma \vdash \{\!|\, \tau_1, \ldots, \tau_n \,|\!\} \preceq \sigma :: \kappa$ *then for some* $m \in 0..n$ *we have* $\sigma = \{\!|\, \tau_1, \ldots, \tau_m \,|\!\}$.
(6) *If* $\Gamma \vdash @\{\!|\, \tau_1, \ldots, \tau_n \,|\!\} \preceq \sigma :: \kappa$ *then either* $\sigma = @\{\!|\, \tau_1, \ldots, \tau_n \,|\!\}$ *or else for some* $m \in 0..n$ *we have* $\sigma = \{\!|\, \tau_1, \ldots, \tau_m \,|\!\}$.

PROOF. Parts 1–4 each follow by induction on the proofs of the subtyping premise. Parts 5 and 6 follow by simultaneous induction on n, using parts 1 and 2. □

LEMMA 9. *(Inversion of Typing)*

(1) *If* $\vdash \mathsf{obj}\ s.\{\!|e_1{:}\tau_1, \ldots, e_n{:}\tau_n|\!\} : \sigma$ *then* $\sigma = @\{\!|\, \tau_1, \ldots, \tau_n \,|\!\}$ *or there exists* $m \in 0..n$ *such that* $\sigma = \{\!|\, \tau_1, \ldots, \tau_m \,|\!\}$. *Further, for all* $i \in 1..n$ *we have* $s{:}\{\!|\, \tau_1, \ldots, \tau_n \,|\!\} \vdash e_i : \tau_i$.
(2) *If* $\vdash \ulcorner \kappa \urcorner : \sigma$ *then* $\sigma = (@\{\!|\, \tau_1, \ldots, \tau_n \,|\!\}) \Rightarrow \tau_k$ *with* $k \in 1..n$ *or else* $\sigma = \{\!|\, \tau_1, \ldots, \tau_n \,|\!\} \Rightarrow \tau_k$ *with* $k \in 1..n$.
(3) *If* $\vdash (\mathsf{pack}\ \tau_1\ \mathsf{and}\ e\ \mathsf{as}\ \tau_2) : \sigma$ *then* $\sigma = \exists \alpha \preceq \sigma_1 :: \kappa_1 . \sigma_2$ *where* $\vdash \tau_1 \preceq \sigma_1 :: \kappa_1$ *and* $\vdash e : \sigma_2[\tau_1/\alpha]$.

PROOF. By induction on the proofs of the premises, using Lemma 8, and (for part 3) Lemma 5. □

LEMMA 10. *(Canonical Forms)*

(1) *If* $\vdash v : @\tau$ *then* $\tau = \{\!|\, \sigma_1, \ldots, \sigma_n \,|\!\}$ *and* $v = \mathsf{obj}\ s.\{\!|e_1{:}\tau_1, \ldots, e_n{:}\tau_n|\!\}$.
(2) *If* $\vdash v : \{\!|\, \tau_1, \ldots, \tau_n \,|\!\}$ *then* $v = \mathsf{obj}\ s.\{\!|e_1{:}\tau_1, \ldots, e_n{:}\tau_n, e_{n+1}{:}\tau_{n+1}, \ldots, e_{n+m}{:}\tau_{n+m}|\!\}$ *with* $m \geq 0$.
(3) *If* $\vdash v : \tau_1 \Rightarrow \tau_2$ *then* $v = \ulcorner k \urcorner$ *and either* $\tau = \{\!|\, \sigma_1, \ldots, \sigma_n \,|\!\}$ *with* $k \in 1..n$ *and* $\tau_2 = \sigma_k$, *or else* $\tau = @\{\!|\, \sigma_1, \ldots, \sigma_n \,|\!\}$ *with* $k \in 1..n$ *and* $\tau_2 = \sigma_k$.

(4) *If* $\vdash v : \exists\alpha{\preceq}\tau_1{::}\kappa_1.\tau_2$ *then* $v = (\mathsf{pack}\ \sigma_1\ \mathsf{and}\ v'\ \mathsf{as}\ \sigma_2)$ *with* $\vdash \sigma_1 \preceq \tau_1 :: \kappa_1$ *and* $\vdash v' : \tau_2\big[\sigma_1/\alpha\big]$.

PROOF. Each part follows by induction on the proof of the assumption, and cases on the last rule used.

(1) If the proof ends with a use of Rule 31 then the desired result follows by Lemma 7 and induction. Otherwise the proof concludes with Rule 22, in which case the desired result follows trivially.

(2) The only possible case is that the proof ends with a use of Rule 31. By Lemma 7, either the premise of this rule shows that v has an exact type (in which case the desired result follows from part 1) or else it has an inexact object type (in which case the desired result follows by induction).

The remaining parts of the proof follow similarly. □

LEMMA 11. *(Decomposition and Replacement) If* $\Gamma \vdash \mathcal{E}[e] : \tau$ *then there exists a type* σ *such that* $\Gamma \vdash e : \sigma$*, and whenever* $\Gamma \vdash e' : \sigma$ *we have* $\Gamma \vdash \mathcal{E}[e'] : \tau$.

PROOF. By induction on the proof of the assumption. □

Proof of Theorem 1

PROOF. By Lemma 11, it suffices to consider only the cases in which $e \rightsquigarrow e'$ is a primitive step. Also, without loss of generality we may assume that the proof $\vdash e : \tau$ does not end with a use of the subsumption rule (Rule 31).

— Case: $e = (\mathsf{obj}\ s.\{\!|e_1{:}\tau_1, \ldots, e_n{:}\tau_n|\!\}).\ulcorner k \urcorner$ and $e' = e_k\big[\mathsf{obj}\ s.\{\!|e_1{:}\tau_1, \ldots, e_n{:}\tau_n|\!\}/s\big]$. By inversion of Rule 23 there exists a type σ such that $\vdash \mathsf{obj}\ s.\{\!|e_1{:}\tau_1, \ldots, e_n{:}\tau_n|\!\} : \sigma$ and $\vdash \ulcorner \kappa \urcorner : \sigma{\Rightarrow}\tau$. By Lemma 9, we have that $\tau = \tau_k$ and for all $i \in 1..n$, $s{:}\{\!|\, \tau_1, \ldots, \tau_n \,|\!\} \vdash e_i : \tau_i$. Thus by Rules 22 and 31 $\vdash \mathsf{obj}\ s.\{\!|e_1{:}\tau_1, \ldots, e_n{:}\tau_n|\!\} : \{\!|\, \tau_1, \ldots, \tau_n \,|\!\}$ and so by Lemma 5 we have $\vdash e_k\big[\mathsf{obj}\ s.\{\!|e_1{:}\tau_1, \ldots, e_n{:}\tau_n|\!\}/s\big] : \tau_k$ as desired.

— Case: $e = (\mathsf{obj}\ s.\{\!|e_1{:}\tau_1, \ldots, e_n{:}\tau_n|\!\}){\leftarrow\!\!+}(s){=}e_{n+1}{:}\tau_{n+1}$, and $e' = \mathsf{obj}\ s.\{\!|e_1{:}\tau_1, \ldots, e_{n+1}{:}\tau_{n+1}|\!\}$. By inversion of Rule 24, Lemma 3, and Lemma 6, we have that $\tau = @\{\!|\, \tau_1, \ldots\ \tau_{n+1} \,|\!\}$. By inversion of Rule 24 and Lemma 9 we have that $s{:}\{\!|\, \tau_1, \ldots, \tau_n \,|\!\} \vdash e_i : \tau_i$ for all $i \in 1..n$, and that $s{:}\{\!|\, \tau_1, \ldots, \tau_{n+1} \,|\!\} \vdash e_{n+1} : \tau_{n+1}$. By Lemma 4, for all $i \in 1..(n{+}1)$ we have $s{:}\{\!|\, \tau_1, \ldots, \tau_{n+1} \,|\!\} \vdash e_i : \tau_i$, and hence $\vdash \mathsf{obj}\ s.\{\!|e_1{:}\tau_1, \ldots, e_{n+1}{:}\tau_{n+1}|\!\} : @\{\!|\, \tau_1, \ldots, \tau_{n+1} \,|\!\}$ as desired.

— Case: $e = (\mathsf{obj}\ s.\{\!|e_1{:}\tau_1, \ldots, e_n{:}\tau_n|\!\}){\leftarrow}\ulcorner k \urcorner(s){=}e'_k$ and $e' = \mathsf{obj}\ s.\{\!|e_1{:}\tau_1, \ldots, e_{k-1}{:}\tau_{k-1}, e'_k{:}\tau_k, e_{k+1}{:}\tau_{k+1}, \ldots, e_n{:}\tau_n|\!\}$. By inversion of Rule 25 there exist types σ_k and σ' such that $\vdash \mathsf{obj}\ s.\{\!|e_1{:}\tau_1, \ldots, e_n{:}\tau_n|\!\} : \tau$, that $\vdash \ulcorner k \urcorner : \tau{\Rightarrow}\sigma_k$, that $\vdash \tau \preceq \sigma' :: \mathsf{Ty}$, that $\vdash \sigma' :: \mathsf{In}$, and that $s{:}\sigma' \vdash e_k :: \sigma_k$. By Lemma 9 there are two cases: either $\tau = @\{\!|\, \tau_1, \ldots, \tau_p \,|\!\}$ with $p = n$ or $\tau = \{\!|\, \tau_1, \ldots, \tau_p \,|\!\}$ with $p \in 1..n$. Either way, by Lemma 9 we have that $k \in 1..p$ and $\sigma_k = \tau_k$. By Lemmas 8 and 6 we know $\sigma' = \{\!|\, \tau_1, \ldots, \tau_m \,|\!\}$ for some $m \in 0..p$. Then by Lemma 4 we have $s{:}\{\!|\, \tau_1, \ldots, \tau_n \,|\!\} \vdash e'_k : \tau_k$, so by Lemma 9 and Rule 22 and (if τ is inexact) Rule 31, we have $\vdash e' : \tau$ as required.

— Case: $e = \mathsf{next\_index}(@\{\!|\, \tau_1, \ldots, \tau_n \,|\!\})$ and $e' = \ulcorner n{+}1 \urcorner$. By inversion of Rule 26 and Lemma 6, we have $\tau = \{\!|\, \tau_1, \ldots, \tau_{n+1} \,|\!\} \Rightarrow \tau_{n+1}$ for some type τ_{n+1}. By Rule 27 we have $\vdash \ulcorner n{+}1 \urcorner : \{\!|\, \tau_1, \ldots, \tau_{n+1} \,|\!\} \Rightarrow \tau_{n+1}$, as desired.

— Case: $e = \mathsf{open}\ (\mathsf{pack}\ \tau'\ \mathsf{and}\ v\ \mathsf{as}\ \sigma)\ \mathsf{as}\ \alpha\ \mathsf{and}\ x\ \mathsf{in}\ e_1$ and $e' = e_1\big[\tau'/\alpha\big]\big[v/x\big]$. By inversion of Rule 30 we have $\vdash \mathsf{pack}\ \tau'\ \mathsf{and}\ v\ \mathsf{as}\ \sigma : \exists \alpha \preceq \tau_1 :: \kappa . \tau_2$, $\alpha \preceq \tau_1 :: \kappa, x{:}\tau_2 \vdash e_1 : \tau$, and that α is not free in τ. By Lemma 9, $\vdash \tau' \preceq \tau_1 :: \kappa$ and $\vdash v : \tau_2\big[\tau'/\alpha\big]$. By Lemma 5 applied twice, and using the fact that α is not free in τ, we have $\vdash e_1\big[\tau'/\alpha\big]\big[v/x\big] : \tau$, as desired.

□

Proof of Theorem 2

Proof. By induction on the proof of $\vdash e : \tau$, and cases on the rule used to prove the conclusion.

— Case: Rules 22, 27, or 29. Then the given expression is a value.

— Case: Rule 23, so that $e = e_1.e_2$. By the inductive hypothesis, either $e_1 \rightsquigarrow e_1'$, $e_2 \rightsquigarrow e_2'$, or both are values. In the first case, we have $e_1.e_2 \rightsquigarrow e_1'.e_2$. If e_1 is a value but e_2 is not, then $e_1.e_2 \rightsquigarrow e_1.e_2'$. Finally, if both are values then by Lemma 10, $e_1 = \mathsf{obj}\ s.\{\!|e_1'{:}\tau_1', \ldots, e_n'{:}\tau_n'|\!\}$ and $e_2 = \ulcorner \kappa \urcorner$. By Lemma 9 we have $k \in 1..n$, so $e_1.e_2 \rightsquigarrow e_k'\big[e_1/s\big]$.

— Case: Rule 24, so that $e = e_1{\leftarrow\!+}(s){=}e_2{:}\tau_2$. By the inductive hypothesis, either $e_1 \rightsquigarrow e_1'$ or e_1 is a value. In the former case, we have $e_1{\leftarrow\!+}(s){=}e_2{:}\tau_2 \rightsquigarrow e_1'{\leftarrow\!+}(s){=}e_2{:}\tau_2$. Otherwise, by Lemma 6 and Lemma 10, $e_1 = \mathsf{obj}\ s.\{\!|e_1'{:}\tau_1', \ldots, e_n'{:}\tau_n'|\!\}$ for some $n \geq 0$. Thus $e_1{\leftarrow\!+}(s){=}e_2{:}\tau_2 \rightsquigarrow \mathsf{obj}\ s.\{\!|e_1'{:}\tau_1', \ldots, e_n'{:}\tau_n', e_2{:}\tau_2|\!\}$.

— Case: Rule 25, so that $e = e_1{\leftarrow}e_2(s){=}e_3$. By the inductive hypothesis, either $e_1 \rightsquigarrow e_1'$, $e_2 \rightsquigarrow e_2'$, or both are values. In the first case, we have $e_1{\leftarrow}e_2(s){=}e_3 \rightsquigarrow e_1'{\leftarrow}e_2(s){=}e_3$. If e_1 is a value but e_2 is not, then $e_1{\leftarrow}e_2(s){=}e_3 \rightsquigarrow e_1{\leftarrow}e_2'(s){=}e_3$. Finally, if both e_1 and e_2 are values then by Lemmas 6 and 7, and Lemma 10 we have $e_1 = \mathsf{obj}\ s.\{\!|e_1'{:}\tau_1', \ldots, e_n'{:}\tau_n'|\!\}$ and $e_2 = \ulcorner \kappa \urcorner$. By Lemma 9 we have $k \in 1..n$, so $e_1{\leftarrow}e_2(s){=}e_3 \rightsquigarrow \mathsf{obj}\ s.\{\!|e_1'{:}\tau_1', \ldots, e_{k-1}'{:}\tau_{k-1}', e_3{:}\tau_k', e_{k+1}'{:}\tau_{k+1}', \ldots, e_n'{:}\tau_n'|\!\}$.

— Case: Rule 26, where $e = \mathsf{next\_index}(\tau_1)$ and $\vdash \tau_1 :: \mathsf{In}$. By Lemma 6, $\tau_1 = \{\!|\, \tau_1', \ldots, \tau_n' \,|\!\}$, so $\mathsf{next\_index}(\tau_1) \rightsquigarrow \ulcorner n + 1 \urcorner$.

— Case: Rule 28. Cannot occur, since the typing environment was assumed to be empty.

— Case: Rule 30, so that $e = \mathsf{open}\ e_1\ \mathsf{as}\ \alpha\ \mathsf{and}\ x\ \mathsf{in}\ e_2$. By the inductive hypothesis, either $e_1 \rightsquigarrow e_1'$, in which case $e \rightsquigarrow \mathsf{open}\ e_1'\ \mathsf{as}\ \alpha\ \mathsf{and}\ x\ \mathsf{in}\ e_2$, or else e_1 is a value. In the latter case by Lemma 10, $e_1 = \mathsf{pack}\ \tau_1'\ \mathsf{and}\ v_1'\ \mathsf{as}\ \tau_2'$, so that $e \rightsquigarrow e_2\big[\tau_1'/\alpha\big]\big[v_1'/x\big]$.

— Case: Rule 31. Follows immediately by the inductive hypothesis.

□