Information and Computation **172**, 2–28 (2002)
doi:10.1006/inco.2000.2925, available online at http://www.idealibrary.com on IDEAL®

Privacy via Subsumption

Jon G. Riecke

Bell Laboratories, Lucent Technologies, 700 Mountain Avenue, Murray Hill, New Jersey 07974

and

Christopher A. Stone[1]

School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, Pennsylvania 15213

Received June 26, 1998

We describe an object calculus allowing object extension and structural subtyping. Each object has a "dictionary" to mediate the connection between names and components. This extra indirection yields the first object calculus combining both object extension and full width subtyping in a type-safe manner. If class inheritance is modeled with object extension, private fields and methods can be achieved directly by scoping restrictions: private fields or methods are those hidden by subsumption. We prove that the type system is sound, discuss a variant allowing covariant self types, and give some examples of the expressiveness of the calculus. © 2002 Elsevier Science

1. INTRODUCTION

One of the most important principles of software engineering is information hiding: the ability to build data or procedural abstractions in order to make programs more readable and maintainable. Object-oriented programming languages provide a number of primitives for information hiding. For example, subsumption can restrict a client of an object to see only the relevant parts: a client expecting fewer methods than an object actually contains need not be aware of (and in most cases cannot invoke) the unknown methods and fields. Class-based languages like C++ and Java have another mechanism for information hiding, namely private annotations on methods and fields. Private methods and fields may be accessed only by other methods defined within the same class.

These two forms of information hiding—subtyping and privacy—appear to be related. In this paper we give an elementary, unified account of both features, using an object calculus [1, 22] to condense the concepts as much as possible. The primitives of our calculus also include object extension, method override, and arbitrary width subtyping and subsumption (i.e., objects with more methods can always be used in contexts expecting fewer methods). We also include an operation for renaming methods, operations that can also be found in the class system of Eiffel [21]. We prove that our type systems prevent run-time type errors.

The main novelty in the calculus is a separation between the components of objects and the names by which they are accessed. Fields and methods are given unique but otherwise arbitrary internal labels; a separate "dictionary" maps external names to the appropriate internal label. (Common implementations of object-oriented languages already use a very similar idea: the internal names are simply offsets within the object or method table, and the mapping from names to offsets is kept separately.) A component whose external name has been lost (either through an explicit operation or implicitly by subsumption) cannot be overridden; other methods refer to this method by its internal name and so are unaffected by the addition of new methods, whatever their external name.

The distinction between internal and external names allows us to avoid a well-known conflict between object extension and width subtyping. For instance, suppose we define an object p by the expression

$$\mathsf{obj}\, s.\{\!|x \triangleright 3 : \mathsf{Int}, getx \triangleright s.x : \mathsf{Int}|\!\}.$$

[1]The second author was sponsored in part by the Advanced Research Projects Agency CSTO under the title "The Fox Project: Advanced Languages for Systems Software," ARPA Order No. C533, issued by ESC/ENS under Contract F19628-95-C-0050.

0890-5401/02 $35.00

The variable s stands for "self," and is dynamically bound to the object upon method invocation. The object p contains two methods, x which returns 3, and *getx* which returns the value of the x method. The type of p is $\{\!|x : \mathsf{Int}, getx : \mathsf{Int}|\!\}$. If we allow width subtyping, p can also have the less precise type $\{\!|getx : \mathsf{Int}|\!\}$. At this type, there is no reason to prevent the addition of a new method x returning the value True of type Bool. In the dynamic semantics of [12, 20] where names are closely connected with components and an object can have at most one x component at a time, this would override the earlier x method and cause *getx* to thereafter return the value True despite statically having type Int. Width subtyping then permits a program that treats a boolean value as an integer—a dynamic type error.

To avoid such errors, the type systems of [13, 20] do not allow full width subtyping: either methods may not be hidden at all, or components can be made "inaccessible", i.e., they are visible and overridable but cannot be invoked. These mechanisms do prevent two methods with the same name being added to the same object, but are unsatisfactory from a software-engineering standpoint: they require the implementor to expose implementation details in interfaces, e.g., names and types of private fields and methods. Moreover, when we use these calculi as a basis for classes, additions or changes to the private methods of a base class may require subclasses to be retypechecked and possibly recompiled. In C++, this forced recompilation is called the "fragile base class" problem [18]. In the worst case, derived classes become ill-formed and large pieces of code must be rewritten.

Our solution is simple: instead of weakening subtyping, we change the dynamic semantics. For instance, when the above example has been translated into our system, the *getx* method refers to x via an internal name so that *getx* continues to return 3, even when the new x method is added. Because the semantics of object extension gives the new method a new internal name, the *getx* method remains unaffected. This forgetting of a method also allows us to give a meaning to private methods, leading us to adopt the slogan of "privacy via subsumption" for the calculus. There is another benefit to this change in dynamic semantics: when using our calculus as a basis for class-based languages, private methods can be changed in arbitrary ways without requiring subclasses to be retypechecked or recompiled. In other words, we can avoid the fragile base class problem.

2. FIRST-ORDER EXTENSIBLE OBJECTS

We begin with a first-order calculus in the sense of [1], i.e., the calculus without a notion of self type. For simplicity, we limit the calculus to a simple delegation-based system; the ability to extend objects allows an elegant encoding of classes. Variants of the calculus have been carefully studied before (e.g., [11–14, 20]). To keep the setting simple, all objects are immutable and objects have no fields; fields can be encoded as methods which ignore their self argument.

TABLE 1

Syntax of the First-Order System

$\tau ::= b$	base type
$\mid (\tau \to \tau')$	function type
$\mid \{\!\|l : \tau_l \ ^{l \in I}\|\!\}$	object type
$\Gamma ::= \bullet$	typing contexts
$\mid \Gamma, x{:}\tau$	
$v ::= c$	constant
$\mid x, y, s, o, \ldots$	variables
$\mid (\lambda x{:}\tau.e)$	abstraction
$\mid \mathsf{obj}\, s.\{\!\|l \triangleright e_l : \tau_l \ ^{l \in I}\|\!\}_\varphi$	object
$e ::= v$	value
$\mid (e\ e')$	function application
$\mid e.l$	method invocation
$\mid e@\varphi$	object renaming
$\mid e \leftarrow\!\!+ l(s) = e' : \tau$	object extension
$\mid e \leftarrow l(s) = e'$	method override

2.1. *Syntax and Static Semantics*

The language, whose syntax appears in Table 1, derives largely from the object calculi of Abadi and Cardelli [1], Fisher, Honsell, and Mitchell [12], and Liquori [20]. We include the standard lambda calculus primitives to avoid unnatural encodings [1]. The types of the language include base types, function types, and object types, where object types draw their method names from an infinite set of labels. Object types only mention the names by which methods are accessed (which must be distinct) and the corresponding return types.

A *dictionary* φ is a finite partial function from labels to labels. Each object contains a dictionary mapping external names to internal names. For instance, the object

$$\mathsf{obj}\, s.\{\!|m \triangleright 3 : \mathsf{Int}, n \triangleright 4 : \mathsf{Int}|\!\}_{[x \mapsto m]}$$

has the dictionary $[x \mapsto m]$; when x is invoked, the actual code invoked is the method internally labeled by m. Note that the code corresponding to the internal label n has no external name, and so cannot be invoked. We use $\varphi(l)$ to denote the application of a dictionary to label l, $(\varphi \circ \varphi')$ to denote the ordinary functional composition of dictionaries, $\varphi[l \mapsto n]$ to denote the partial function that behaves exactly as φ except for mapping l to n, and $id(S)$ to denote the identity function on a set of labels S.

There are three primitive operations on objects besides method invocation. The operation $e@\varphi'$, alters the existing dictionary on an object: it evaluates e to an object and composes φ' with its internal dictionary. In addition to renaming components, this operation can contract the number of methods visible in the object when the range of φ' is smaller than the domain of the dictionary on the object. For example,

$$\mathsf{obj}\, s.\{\!|l_1 \triangleright 3 : \mathsf{Int}, l_2 \triangleright s.l_1 : \mathsf{Int}|\!\}_{\varphi}@[\mathit{getx} \mapsto \mathit{getx}]$$

where $\varphi = [x \mapsto l_1, \mathit{getx} \mapsto l_2]$, evaluates to an object whose only visible method is *getx* with dictionary $[\mathit{getx} \mapsto l_2]$. Similarly, one can increase the number of visible fields by mapping several external labels to the same internal label. (In this case, if one of these methods is overridden multiple methods will appear to change.) The other two operations add or change the methods of objects. An existing method can be replaced within an object by the operation $e_0 \Leftarrow l(s) = e$. The operation $e_0 \Leftarrow\!\!+ l(s) = e : \tau$ adds a new method l to the object denoted by e_0. The method expects a self parameter s, and when invoked evaluates the body e of type τ. Because we do not have depth subtyping (it is unsound in the presence of the override operation), the new method in the extension operation is given an explicit type annotation so that all expressions will have most-specific types. Override does not change the type of an object, so no annotation is needed there.

We identify object expressions or types differing only in the order of their components, and expressions up to renaming of bound variables. Object expressions as well as lambda expressions bind variables. For instance, in $\mathsf{obj}\, s.\{\!|l \triangleright e_l : \tau_l{}^{l \in I}|\!\}_{\varphi}$, s is a bound variable whose scope includes all the method bodies. Similarly, s is bound in the new method body e' in the extension and override operations shown in Table 1.

The static semantics of the language is given in Appendix A. The novel aspects are the subsumption rule for naïve width subtyping and the treatment of dictionaries. The rules use $\mathsf{FV}(\Gamma)$ to denote the variables occurring in the context Γ.

2.2. *Dynamic Semantics*

To give dynamic semantics to the language, we use Felleisen's "evaluation context" formulation [10] of Plotkin's SOS [24]. The syntax of evaluation contexts (a subset of those expressions containing a single hole, denoted $\bullet$) is given by the grammar

$$\begin{aligned} E ::= {} & \bullet \mid (E\ e) \mid (v\ E) \\ & \mid E.l \\ & \mid E@\varphi \\ & \mid E \Leftarrow l(s) = e' \\ & \mid E \Leftarrow\!\!+ l(s) = e' : \tau' \end{aligned}$$

We write $E[e]$ to denote the evaluation context E with the hole replaced by e. The local reduction relation $\rightsquigarrow$ is shown in Table 2. These rules use a syntactic substitution operation, written $[s \mapsto e]\, e'$,

TABLE 2

Local Reduction Steps of First-Order System

$(\lambda x{:}\tau.e)\ v$	$\rightsquigarrow$	$[x \mapsto v]\,e$
$(\mathsf{obj}\,s.\{\!\|m \rhd e_m : \tau_m\ ^{m\in I}\|\!\}_\varphi)@\varphi'$	$\rightsquigarrow$	$\mathsf{obj}\,s.\{\!\|m \rhd e_m : \tau_m\ ^{m\in I}\|\!\}_{\varphi\circ\varphi'}$
$(\mathsf{obj}\,s.\{\!\|m \rhd e_m : \tau_m\ ^{m\in I}\|\!\}_\varphi).l$	$\rightsquigarrow$	$[s \mapsto \mathsf{obj}\,s.\{\!\|m \rhd e_m : \tau_m\ ^{m\in I}\|\!\}_{id(I)}]\,e_{\varphi(l)}$
$(\mathsf{obj}\,s.\{\!\|m \rhd e_m : \tau_m\ ^{m\in I}\|\!\}_\varphi) \leftarrow l(s)=e$	$\rightsquigarrow$	$\mathsf{obj}\,s.\{\!\|m \rhd e_m : \tau_m\ ^{m\in I\setminus\varphi(l)},\ \varphi(l) \rhd [s \mapsto s@\varphi]\,e : \tau_{\varphi(l)}\|\!\}_\varphi$
$(\mathsf{obj}\,s.\{\!\|m \rhd e_m : \tau_m\ ^{m\in I}\|\!\}_\varphi) \leftarrow\!\!+ l(s)=e : \tau$	$\rightsquigarrow$	$\mathsf{obj}\,s.\{\!\|m \rhd e_m : \tau_m\ ^{m\in I},\ l' \rhd [s \mapsto s@\varphi']\,e : \tau\|\!\}_{\varphi'}$
		$\varphi' = \varphi[l \mapsto l']$ where $l' = \mathsf{Fresh}(I)$

which denotes the capture-free substitution of e for s in e'. The function Fresh, given a set of labels, deterministically chooses a new label not in that set.

The most interesting operational rules are the rules for method invocation, override, and extension. During method invocation, the dictionary is stripped and replaced by an identity dictionary. During method extension and override, the new method body is modified with the object's current dictionary; upon invocation it will restore this dictionary to the stripped self argument. The combination of these two features gives method bodies an unchanging view of the object, even though arbitrary changes to the object's dictionary may happen later through other renamings or extensions.

The relation in Table 2 is extended to a one-step evaluation relation on programs: $e \rightsquigarrow e'$ iff there are terms e_1, e_2 such that $e = E[e_1]$, $e_1 \rightsquigarrow e_2$, and $E[e_2] = e'$. We can prove

PROPOSITION 1. (Determinacy). *The relation $\rightsquigarrow$ is a partial function.*

We use $\rightsquigarrow^*$ to denote the reflexive, transitive closure of $\rightsquigarrow$.

The static semantics and dynamic semantics also agree. The key results are:

1. [Subject Reduction] If $\Gamma \vdash e : \sigma$ and $e \rightsquigarrow e'$ then $\Gamma \vdash e' : \sigma$.
2. [Progress] If $\vdash e : \tau$ then either e is a value or else $e \rightsquigarrow e'$.

The proofs for the first-order system appear in Appendix C.

2.3. *Examples*

The first example shows the behavior of the operational semantics. Let $\varphi = [F \mapsto l_1, M \mapsto l_2]$ and define the explicit subtyping coercion $o{:}{>}\tau$ as shorthand for the term $((\lambda x{:}\tau.x)\ o)$. Consider the terms

$$o := (\mathsf{obj}\,s \cdot \{\!\|\}\!\|_{[]})$$

$$o_0 := (o \leftarrow\!\!+ F(s)=5 : \mathsf{Int}) \leftarrow\!\!+ M(s)=(s.F+1) : \mathsf{Int}$$

$$o_0 : \{\!\|F : \mathsf{Int}, M : \mathsf{Int}\|\!\}$$
$$o_0 \rightsquigarrow^* \mathsf{obj}\,s.\{\!\|l_1 \rhd 5 : \mathsf{Int}, l_2 \rhd (s@\varphi).F+1 : \mathsf{Int}\|\!\}_\varphi$$

$$o_1 := o_0 \leftarrow F(s)=7$$

$$o_1 : \{\!\|F : \mathsf{Int}, M : \mathsf{Int}\|\!\}$$
$$o_1 \rightsquigarrow^* \mathsf{obj}\,s.\{\!\|l_1 \rhd 7 : \mathsf{Int}, l_2 \rhd (s@\varphi).F+1 : \mathsf{Int}\|\!\}_\varphi$$

$$o_2 := o_1{:}{>}\{\!\|M : \mathsf{Int}\|\!\}$$

$$o_2 : \{\!\|M : \mathsf{Int}\|\!\}$$
$$o_2 \rightsquigarrow^* \mathsf{obj}\,s.\{\!\|l_1 \rhd 7 : \mathsf{Int}, l_2 \rhd (s@\varphi).F+1 : \mathsf{Int}\|\!\}_\varphi$$

$$o_3 := o_2 \leftarrow\!\!+ F(s)=\mathsf{True} : \mathsf{Bool}$$

$$o_3 : \{\!\|F : \mathsf{Bool}, M : \mathsf{Int}\|\!\}$$
$$o_3 \rightsquigarrow^* \mathsf{obj}\,s.\{\!\|l_1 \rhd 7 : \mathsf{Int}, l_2 \rhd (s@\varphi).F+1 : \mathsf{Int},$$
$$l_3 \rhd \mathsf{True} : \mathsf{Bool}\|\!\}_{[F \mapsto l_3, M \mapsto l_2]}$$

Here o_0 has methods F and M. When a method is invoked, the self parameter s is replaced with an object with an identity dictionary. Thus, it is easy to see that $o_0.F$ evaluates to 5 and $o_0.M$ to 6. In o_1 we override F with a method that returns 7; $o_1.F$ evaluates to 7 and $o_1.M$ to 8. To obtain o_2, we use subsumption on o to make method F private, leaving only one visible method M. Then $o_2.M$ still evaluates to 8. The type system would reject any attempt to *override* F in o_2, since o_2 has no visible F method. It is legal, however, to *extend* o_2 to o_3 by adding a new method called F (which here happens to return a boolean value). The previous F method is still present in the underlying object, and evaluating $o_3.M$ still gives 8, while $o_3.F$ returns True.

As this example shows, extending an object never changes the behavior of pre-existing methods. When a method is added to an object, we arrange for its body to invoke methods in self using internal labels. Its behavior does not change unless one of these is overridden, which cannot occur unless there is a corresponding external label.

This example also raises another point: object extension must be used carefully. One may always use extension in place of method override, but the consequences are different. For instance, consider the term

$$o_4 := o_0 \leftarrow\!\!+ F(s) = 7 : \mathsf{Int}$$

which resembles o_1 except that we use extension rather than override. The term is typable because the object o is implicitly forced (via subsumption) to have an object type with only one method M. As such, $o_4.M$ returns 6 while $o_1.M$ returns 8. The programmer must be careful to determine which of these behaviors is correct and use the appropriate operation.

For a similar example, define the function *getf by*

$$\mathit{getf} := \lambda p{:}(\{\!|F : \mathsf{Int}|\!\}).(p.F)$$

Then define the objects

$$\begin{aligned} p_1 := {} & \mathsf{obj}\, s.\{\!|\,|\!\}_{[]} \\ & \leftarrow\!\!+ F(s) = 4 : \mathsf{Int} \\ & \leftarrow\!\!+ M_1(s) = s.F : \mathsf{Int} \\ & \leftarrow\!\!+ M_2(s) = \mathit{getf}(s) : \mathsf{Int} \\ p_2 := {} & p_1 \\ & \leftarrow\!\!+ F(s) = 5 : \mathsf{Int} \\ & \leftarrow\!\!+ N_1(s) = s.F : \mathsf{Int} \\ & \leftarrow\!\!+ N_2(s) = \mathit{getf}(s) : \mathsf{Int} \end{aligned}$$

Then

$$\begin{aligned} p_1 &: \{\!|F : \mathsf{Int}, M_1 : \mathsf{Int}, M_2 : \mathsf{Int}|\!\} \\ p_2 &: \{\!|F : \mathsf{Int}, M_1 : \mathsf{Int}, M_2 : \mathsf{Int}, N_1 : \mathsf{Int}, N_2 : \mathsf{Int}|\!\} \end{aligned}$$

$$\begin{array}{ll} p_1.F \rightsquigarrow^* 4 & p_2.F \rightsquigarrow^* 5 \\ p_1.M_1 \rightsquigarrow^* 4 & p_2.M_1 \rightsquigarrow^* 4 \\ p_1.M_2 \rightsquigarrow^* 4 & p_2.M_2 \rightsquigarrow^* 4 \\ & p_2.N_1 \rightsquigarrow^* 5 \\ & p_2.N_2 \rightsquigarrow^* 5 \end{array}$$

Although $p_2.M_1$ and $p_2.N_1$ may appear to have the same code, they evaluate to different values because—just as in the preceding series of examples—these two methods refer to different object components by the name F. Slightly less obviously, the same effect occurs in $p_2.M_2$ and $p_2.N_2$; although both methods believe that the self object has a method F returning an integer, they disagree on which component within the self object is that F method; dynamically the two methods will pass s to the *getf* function with different dictionaries attached, which causes the *getf* calls to return different results. In the first-order system, this dictionary manipulation is hidden in the dynamic semantics. In the

second-order system of the next section, we are forced to make such manipulations explicit and hence will revisit this example.

For yet another example, consider the term

$$\lambda p{:}\{\!|x : \mathsf{Int}|\!\}.(p \leftarrow\!\!+ getx(s) = s.x : \mathsf{Int})$$

This function can be given the type

$$(\{\!|x : \mathsf{Int}|\!\} \to \{\!|x : \mathsf{Int}, getx : \mathsf{Int}|\!\}).$$

In contrast to other formalisms, this function may be applied to *any* object with an x method of type Int, regardless of its other methods. On the other hand, there is some information loss: if we apply this function to an object with (public) methods x, y, and z, the result has just two public methods x and *getx*; y and z are hidden in the act of subsumption. One would need an extension such as row variables [28] or bounded polymorphism [8] in order to avoid this behavior.

Classes can also be encoded in the system, where a class provides a way to create objects and to inherit from the class. We encode classes as object-generating functions. This means that classes have a single constructor function, as in Objective Caml [26]; more complex encodings with multiple constructor functions are possible.

A very standard example of classes involves classes for "points" and "colored points." The public types of points and colored points are

$$PT := \{\!|getx : \mathsf{Int}|\!\}$$
$$CPT := \{\!|getx : \mathsf{Int}, getc : \mathsf{Color}|\!\}.$$

The classes are defined by

$$\begin{aligned} pt\_class := \lambda(x_0 : \mathsf{Int}).\ & \\ & (\mathsf{obj}\, s.\{\!|\}\!|_{[]} \\ & \leftarrow\!\!+ x(s) = x_0 : \mathsf{Int} \\ & \leftarrow\!\!+ getx(s) = s.x : \mathsf{Int} \\ &) :> PT \\ cpt\_class := \lambda(x_0{:}\mathsf{Int}).\ \lambda(c_0{:}\mathsf{Color}).\ & \\ & (pt\_class(x_0) \\ & \leftarrow\!\!+ c(s) = c_0 : \mathsf{Color} \\ & \leftarrow\!\!+ getc(s) = s.c : \mathsf{Int}) :> CPT \end{aligned}$$

Note that objects created by *pt_class* have a field x which is used by *getx*, but will be hidden from external view by subsumption. Clients can invoke this class to create point objects, but by the static typing they cannot directly access the x component. Furthermore, the function *cpt_class* inherits from the point class, but the color-point class methods also cannot invoke or override the x component. We have added a private field c to the class of colored points, accessible only by the *getc* method. To typecheck and compile *cpt_class*, we need only know the type $(\mathsf{Int} \to PT)$ of *pt_class*, which does not mention x. The *cpt_class* function could choose to add a method named x of any type, which would not interfere with the private field x inherited from *pt_class*.

We could expand on this example to encode *protected* components (fields and methods only visible to subclasses). In this case, a class becomes two functions, one to be invoked by subclasses and the other to be invoked by clients. The first function generates the object and restricts it to a "protected" interface, hiding the private components. The second function further restricts the type of the object to expose only the public components. This "protection via subtyping" encoding has been discussed elsewhere [1, 15].

3. SECOND-ORDER EXTENSIBLE OBJECTS

In a calculus of immutable objects, it is natural to consider objects that can return updated copies of themselves. For example, we might define a type of movable points, which could be defined (using a recursive type definition) as

$$MPT' := \{\!|getx : \mathsf{Int}, move : (\mathsf{Int} \to MPT')|\!\}$$

where the *move* operation takes an amount to offset the position of the returned point. Now suppose we extend $pt' : MPT'$ to a colored point by adding a *getc* method returning a color. The resulting object would have type

$$MCPT' := \{\!|getx : \mathsf{Int}, move : (\mathsf{Int} \to MPT'), getc : \mathsf{color}|\!\}$$

Unfortunately, if $cpt' : MCPT'$ then $cpt'.move$ is a function which still returns a value of type MPT'; the color is lost.

A "second-order calculus," in the parlance of [1], can repair the problem. In a second-order calculus, method types can refer to "the type of the object whose method is being invoked". This type is usually called a "self type". When the object is extended, the self type changes correspondingly. Thus we define

$$MPT := \mathsf{Obj}\,\alpha.\{\!|getx : \mathsf{Int}, move : (\mathsf{Int} \to \alpha)|\!\}$$

where α represents the type of self, and is bound within the object type. Then the extension to add a color would have type

$$MCPT := \mathsf{Obj}\,\alpha.\{\!|getx : \mathsf{Int}, move : (\mathsf{Int} \to \alpha), getc : \mathsf{color}|\!\}$$

Assuming $pt : MPT$ and $cpt : MCPT$, the method invocation $pt.move$ has type $[\alpha \mapsto MPT]\ (\mathsf{Int} \to \alpha) = (\mathsf{Int} \to MPT)$, and the method invocation $cpt.move$ has type $[\alpha \mapsto MCPT]\,(\mathsf{Int} \to \alpha) = (\mathsf{Int} \to MCPT)$ as desired.

Because our objects carry dictionaries, there is a complication. In the above example, *move* returns an updated version of the self object, with the same type as the point being moved. The operational semantics for method invocation, however, *discards* the dictionary attached to the object and replaces it with the identity dictionary. Furthermore, the code for *move* must work in all extensions and renamings of the object; there is no static means of determining what the dictionary will be when *move* is invoked.

The solution is to make dictionary manipulations much more explicit. Dictionaries become values, and method invocation involves binding a dictionary as well as self. Using that mechanism, a method can reattach a dictionary to the self object. To preserve typing information, the method invocation and method override operations are parameterized by an extra dictionary. This dictionary is used as an extra indirection in specifying the component intended by the given method name; the first-order operations correspond to the case in which this dictionary is the identity mapping. This allows us to locate an object's component using a given dictionary without modifying the dictionary carried by the object.

3.1. *Syntax and Static Semantics*

In the second-order system, dictionaries are values and thus must have types. The type $\tau_1 \Rightarrow \tau_2$ denotes dictionaries that can be used to rename an object of type τ_1 to an object of type τ_2. We distinguish dictionary types from function types because only dictionaries (and variables bound to dictionaries) may appear as the extra parameter for method invocation and method override. A dictionary φ of type $\tau_1 \Rightarrow \tau_2$ can be coerced to a function $(\lambda x{:}\tau_1.(x@\varphi))$ of type $(\tau_1 \to \tau_2)$.

As with the first-order system, object values have a binding representing self. In the second-order system, however, self has two types: the external type (usually denoted α here), and the internal type (usually denoted β), which is the type of self during method invocation. This idea is not new; the types α and β closely resemble the `MyType` and `SelfType` constructs from TOOPL [6]. The dictionary (usually denoted d) on an object maps β to α, and is instantiated to the current dictionary during method

invocation. Thus, the syntax of an object in the second-order system is

$$\mathsf{obj}(\alpha, \beta, s, d).\{\!|m \triangleright e_m : \tau_m \ ^{m\in I}|\!\}_{\varphi}$$

where α, β, s, and d are all bound within the body of the object.

In the typing rules for objects, the methods are checked under the assumptions that $d : \beta \Rightarrow \alpha$ and $s : \beta$. We can only guarantee that the dictionary d is correct *for the object at the time the method is invoked*, and which may not be applicable for the current object. All we know statically about β is that it will be a subtype of the current internal representation, i.e., β is a partially abstract type. In the typing rules, β must not appear free in the types of the methods—only α may appear free—which also reflects the concept of an "existential" or abstract type [23].

Similar modifications must be made to the method override and extension operations. These operations are further parameterized by a dictionary d' which is the dictionary *at the time the method is added or overridden*. (Recall that d represents a dictionary in place when the method is later invoked.) Since we do not know statically the internal type of the object being altered, the new method body can assume nothing about β except that it is an object type. What we do know is an object type τ for the object, and that the current dictionary when attached to the object gives the object this type; thus the new method is typechecked under the assumption $d' : \beta \Rightarrow \tau$.

Finally, as noted above both method invocation and method override are additionally parameterized by a value v, which will be a constant dictionary or a variable with a dictionary type.

Table 3 gives the syntax of the second-order system. There is one technical constraint: in object types, the type α of self must appear covariantly inside object types. We say that α appears covariantly in τ if any of the following is true:

- α is not free in τ;
- τ is α;
- τ is $(\tau_1 \to \tau_2)$ or $\tau_1 \Rightarrow \tau_2$, where α appears contravariantly in τ_1 and covariantly in τ_2;

Similarly, α appears contravariantly in τ if any of the following is true:

- α is not free in τ;
- τ is $(\tau_1 \to \tau_2)$ or $\tau_1 \Rightarrow \tau_2$, where α appears covariantly in τ_1 and contravariantly in τ_2;

We would need more restrictive width subtyping to avoid unsoundness if the α were allowed to appear

TABLE 3

Syntax of the Second-Order System

τ	$::= b$	base type
	$\mid \alpha$	type variable
	$\mid (\tau \to \tau')$	function type
	$\mid \tau \Rightarrow \tau'$	dictionary type
	$\mid \mathsf{Obj}\,\alpha.\{\!|l : \tau_l \ ^{l\in I}|\!\}$	object type
Γ	$::= \bullet$	typing contexts
	$\mid \Gamma, x{:}\tau$	
	$\mid \Gamma, \alpha \preceq \tau$	
v	$::= c$	constant
	$\mid x, s, d, \ldots$	variable
	$\mid \lambda x{:}\tau.e$	function
	$\mid \varphi$	dictionary
	$\mid \mathsf{obj}(\alpha, \beta, s, d).\{\!|l \triangleright e_l : \tau_l' \ ^{l\in I}|\!\}_{\varphi}$	object
e	$::= v$	value
	$\mid e_1\ e_2$	application
	$\mid e@v$	object renaming
	$\mid e \cdot_v l$	method invocation
	$\mid e \leftarrow_v l(\alpha, \beta, s, d, d') = e'$	method override
	$\mid e \leftarrow\!\!+ l(\alpha, \beta, s, d, d') = e' : \tau$	object extension

non-covariantly (see [1] for examples). As such, this system does not handle binary methods (see [7] for a thorough discussion).

The static semantics of the second-order calculus appears in Appendix B. The rules use the abbreviation

$$\top := \mathsf{Obj}\,\alpha.\{\!|\,|\!\}$$

for the object type conveying the least information. In addition to the changes discussed above, we must handle type variables and bounded quantification in the typing context. Typing contexts Γ are finite, partial functions from variables to types and from type variables to upper bounds. For instance, the context $\Gamma = (x{:}\tau, \alpha \preceq \tau')$ denotes a typing context with domain x, α, and states that x is assumed to have type τ and α has an upper bound of τ'. The domain of a context Γ is denoted $\mathsf{Dom}(\Gamma)$.

3.2. *Dynamic Semantics*

The dynamic semantics for the second-order calculus uses evaluation contexts of the form

$$\begin{array}{rcl} E & ::= & \bullet \mid (E\ e) \mid (v\ E) \\ & \mid & E@v \\ & \mid & E \cdot_v l \\ & \mid & E \leftarrow_v l(\alpha, \beta, s, d, d') = e' \\ & \mid & E \leftarrow\!\!+ l(\alpha, \beta, s, d, d') = e' : \tau' \end{array}$$

The rules for reducing redexes appear in Table 4, and these rules are extended to a relation on expressions via evaluation contexts in the same way as the first-order system.

As in the first-order system, the difficult rules are the rules for method invocation, override, and extension. In method invocation, the self parameter is replaced by the object with the identity dictionary; the parameter d is bound to the current dictionary. The types α and β are bound to the appropriate types: α matches the external type of the object (with the dictionary φ in place), and β matches the internal type of the object (with the identity dictionary). In method override and extension, d' is bound in the body to the dictionary at the time of the operation. Like in the first-order system, this gives the body the ability to use the object in the way it could be used at the time of override or extension. The self parameter in the body is not changed, however.

As with the first-order system, the static and dynamic semantics agree in the following ways:

1. [Subject Reduction] If $\vdash e : \tau$ and $e \rightsquigarrow e'$, then $\vdash e : \tau$.
2. [Progress] If $\vdash e : \tau$, then e is a value or $e \rightsquigarrow e'$ for some expression e'.

The proofs appear in Appendix D.

TABLE 4

Local Reduction Steps for Second-Order System

$$\begin{array}{lcl}
(\lambda x{:}\tau.e)\ v & \rightsquigarrow & [x \mapsto v]\,e \\
(\mathsf{obj}(\alpha, \beta, s, d).\{\!|m \rhd e_m : \tau_m\ ^{m \in I}|\!\}_\varphi)@\varphi' & \rightsquigarrow & \mathsf{obj}(\alpha, \beta, s, d).\{\!|m \rhd e_m : \tau_m\ ^{m \in I}|\!\}_{\varphi \circ \varphi'} \\
(\mathsf{obj}(\alpha, \beta, s, d).\{\!|m \rhd e_m : \tau_m\ ^{m \in I}|\!\}_\varphi) \cdot_{\varphi'} l & \rightsquigarrow & [d \mapsto \varphi]\,[s \mapsto \mathit{self}]\,[\alpha \mapsto A]\,[\beta \mapsto B]\,e_{\varphi(\varphi'(l))} \\
 & & \text{where} \quad A = \mathsf{Obj}\,\alpha.\{\!|l : \tau_{\varphi(l)}\ ^{l \in \mathsf{Dom}(\varphi)}|\!\} \\
 & & \qquad\quad B = \mathsf{Obj}\,\alpha.\{\!|m : \tau_m\ ^{m \in I}|\!\} \\
 & & \qquad\quad \mathit{self} = \mathsf{obj}(\alpha, \beta, s, d).\{\!|m \rhd e_m : \tau_m\ ^{m \in I}|\!\}_{id(I)} \\
(\mathsf{obj}(\alpha, \beta, s, d).\{\!|m \rhd e_m : \tau_m\ ^{m \in I}|\!\}_\varphi) \leftarrow_{\varphi'} l(\alpha, \beta, s, d, d') = e & & \\
 & \rightsquigarrow & \mathsf{obj}(\alpha, \beta, s, d).\{\!|m \rhd e_m : \tau_m\ ^{m \in I \setminus \varphi(\varphi'(l))},\ \varphi(\varphi'(l)) \rhd [d' \mapsto \varphi]\,e : \tau_{\varphi(\varphi'(l))}|\!\}_\varphi \\
(\mathsf{obj}(\alpha, \beta, s, d).\{\!|m \rhd e_m : \tau_m\ ^{m \in I}|\!\}_\varphi) \leftarrow\!\!+ l(\alpha, \beta, s, d, d') = e : \tau & & \\
 & \rightsquigarrow & \mathsf{obj}(\alpha, \beta, s, d).\{\!|m \rhd e_m : \tau_m\ ^{m \in I},\ l' \rhd [d' \mapsto \varphi']\,e : \tau|\!\}_{\varphi'} \\
 & & \text{where } A = \mathsf{Obj}\,\alpha.\{\!|l : \tau_{\varphi(l)}\ ^{l \in \mathsf{Dom}(\varphi)},\ l{:}\tau|\!\},\ \mathsf{Fresh}(I) = l',\ \varphi' = \varphi[l \mapsto l']
\end{array}$$

3.3. *Examples*

We first revisit the simple examples from Section 2.3. Again let $\varphi = [F \mapsto l_1,\ M \mapsto l_2]$. The equivalent terms are then

$$\begin{aligned} o &:= (\mathsf{obj}(\alpha, \beta, s, d).\{\!|\}\!\}_{[]}) \\ o_0 &:= o \\ &\quad \leftarrow\!\!+ F(\alpha, \beta, s, d, d') = 5 : \mathsf{Int} \\ &\quad \leftarrow\!\!+ M(\alpha, \beta, s, d, d') = (s._{d'} F + 1) : \mathsf{Int} \\ o_1 &:= o_0 \leftarrow_{[F \mapsto F]} F(\alpha, \beta, s, d, d') = 7 \end{aligned}$$

Note that $o_0, o_1 : \mathsf{Obj}\,\alpha.\{\!|F : \mathsf{Int}, M : \mathsf{Int}|\!\}$, and

$$o_0 \rightsquigarrow^* \mathsf{obj}(\alpha, \beta, s, d).\{\!|l_1 \triangleright 5 : \mathsf{Int}, l_2 \triangleright s \cdot_\varphi F + 1 : \mathsf{Int}|\!\}_\varphi$$

$$o_1 \rightsquigarrow^* \mathsf{obj}(\alpha, \beta, s, d).\{\!|l_1 \triangleright 7 : \mathsf{Int}, l_2 \triangleright s \cdot_\varphi F + 1 : \mathsf{Int}|\!\}_\varphi$$

The object o_1 could also be defined as

$$o_1 := o_0 \leftarrow_{[G \mapsto F]} G(\alpha, \beta, s, d, d') = 7$$

which would evaluate to exactly the same object value.

We next recast the *getf* example into the second-order system. We define the function *getf* by

$$getf := \lambda p{:}(\mathsf{Obj}\,\alpha.\{\!|F : \mathsf{Int}|\!\}).(p.F)$$

and the objects p_1 and p_2 by

$$\begin{aligned} p_1 &:= \mathsf{obj}(\alpha, \beta, s, d).\{\!|\}\!\}_{[]} \\ &\quad \leftarrow\!\!+ F(\alpha, \beta, s, d, d') = 4 : \mathsf{Int} \\ &\quad \leftarrow\!\!+ M_1(\alpha, \beta, s, d, d') = s \cdot_{d'} F : \mathsf{Int} \\ &\quad \leftarrow\!\!+ M_2(\alpha, \beta, s, d, d') = getf(s@d') : \mathsf{Int} \\ p_2 &:= p_1 \\ &\quad \leftarrow\!\!+ F(\alpha, \beta, s, d, d') = 5 : \mathsf{Int} \\ &\quad \leftarrow\!\!+ N_1(\alpha, \beta, s, d, d') = s \cdot_{d'} F : \mathsf{Int} \\ &\quad \leftarrow\!\!+ N_2(\alpha, \beta, s, d, d') = getf(s@d') : \mathsf{Int} \end{aligned}$$

Invoking methods in these two objects yields the same integer values as in the first-order example, e.g., $p_1 \cdot_{[F \mapsto F]} F \rightsquigarrow^* 4$. In this presentation, it is more clear that the calls to $getf$ in M_2 and N_2 are passed objects with different dictionaries, as d' is instantiated to different values in the two methods.

The expressions $s \cdot_{d'} F$ in the M_1 and N_1 methods above could have been written in the operationally equivalent form

$$(s@d') \cdot_{[F \mapsto F]} F$$

so that all method invocation operations would be annotated with identity dictionaries, mimicking the behavior of the first-order system. However, there are cases where these two forms have different typing behavior. Consider the object

$$q_1 := \mathsf{obj}(\alpha, \beta, s, d).\{\!|M \triangleright 3 : \mathsf{Int}|\!\}_{[M \mapsto M]}$$

of type $\mathsf{Obj}\,\alpha.\{\!|M : \mathsf{Int}|\!\}$. Then $(q_1@[N \mapsto M]) \cdot_{[N \mapsto N]} N$ and $q_1 \cdot_{[N \mapsto M]} N$ are operationally equivalent; both expressions have type Int and evaluate to 3.

In constrast, for the object

$$q_2 := \mathsf{obj}(\alpha, \beta, s, d).\{\!|M \triangleright s@d : \alpha|\!\}_{[M \mapsto M]}$$

of type $\mathsf{Obj}\,\alpha.\{\!|M : \alpha|\!\}$, the two corresponding expressions are different: $q_{2\cdot[N\mapsto M]}N$ has type $\mathsf{Obj}\,\alpha.\{\!|M : \alpha|\!\}$ and evaluates to q_2, while $(q_2@[N \mapsto M])_{\cdot[N\mapsto N]}N$ has type $\mathsf{Obj}\,\alpha.\{\!|N : \alpha|\!\}$ and evaluates to

$$\mathsf{obj}(\alpha, \beta, s, d).\{\!|M \triangleright s@d : \alpha|\!\}_{[N\mapsto M]}.$$

We now revisit the class example from Section 2.3 to create classes for movable points and colored points.

$$\begin{aligned}
\mathit{mpt_class} := \lambda(x_0 : \mathsf{Int}).\ & \\
& (\mathsf{obj}(\alpha, \beta, s, d).\{\!|\,|\!\}_{[]} \\
& \quad \leftarrow\!\!+ x(\alpha, \beta, s, d, d') = x_0 : \mathsf{Int} \\
& \quad \leftarrow\!\!+ \mathit{getx}(\alpha, \beta, s, d, d') = s\cdot_{d'} x : \mathsf{Int} \\
& \quad \leftarrow\!\!+ \mathit{move}(\alpha, \beta, s, d, d') = \\
& \qquad \lambda y{:}\mathsf{Int}.(\mathsf{let}\ z = s\cdot_{d'} \mathit{getx} \\
& \qquad\qquad \mathsf{in}\ s \leftarrow_{d'} x(\alpha_1, \beta_1, s_1, d_1, d_1') \\
& \qquad\qquad\qquad = z + y) \\
& \qquad @d : (\mathsf{Int} \to \alpha) \\
&) :> \mathit{MPT} \\
\mathit{mcpt_class} := \lambda(x_0{:}\mathsf{Int}).\ \lambda(c_0{:}\mathsf{Color}). & \\
& (\mathit{pt_class}(x_0) \\
& \quad \leftarrow\!\!+ c(\alpha, \beta, s, d, d') = c_0 : \mathsf{Color} \\
& \quad \leftarrow\!\!+ \mathit{getc}(\alpha, \beta, s, d, d') = s\cdot_{d'} c : \mathsf{Color} \\
&) :> \mathit{MCPT}
\end{aligned}$$

In the *move* method, the use of the dictionary d' to parameterize the method override is essential; it allows us to use s as an object value while statically preserving the dictionary attached to s (so that the updated object retains type β and can be coerced to the external self type α).

4. CONCLUSIONS

4.1. *Implementation Issues*

There is a tradeoff in using explicit dictionaries: dictionary manipulation may induce a run-time cost. In a setting where our object calculus is used directly, there are ways for modestly reducing the run-time costs of dictionaries. For example, in compiling the dictionary composition operation $e@\varphi'$, one can either choose to calculate the composition of e's dictionary φ with φ' directly, or calculate the composition *lazily* as the new object gets requests for methods. The former may be more efficient when there are frequent compositions and method invocations, the latter more efficient when there are fewer compositions.

Similarly, though it is certainly unsound to drop components from an object when they are hidden by subsumption, it is possible to drop these components from the *dictionary*. By turning subsumption into a run-time coercion on dictionaries, an implementation can ensure that the order and position of entries in an object's dictionary always matches the static type; then dictionary lookups are guaranteed to take constant time. Whether this is a good idea depends on the frequency of subsumptions, and the cost of searching a dictionary of unknown size.

If one knows more about the style of programming in the calculus, more efficiencies can be gained. For instance, the calculus could be used as a compilation target for single-inheritance class-based languages. In these languages, each class determines a "method table" that can be shared among all objects of the class (the fields of each object, of course, must be maintained separately). The mapping of method names to indices in the method table is the dictionary. Since the method table can be statically determined,

method calls through self need not be matched to a slot in the method table: they can immediately jump to the method. That is, when φ is statically determinable, the compiler can do dictionary lookups at compile-time and not generate code involving this dictionary for $(e@\varphi).l$ in the first-order calculus or $e._\varphi l$ in the second-order calculus. We also know that the self variable s within an object refers to an object with the identity dictionary, so that $s@\varphi$ can be implemented as a dictionary replacement operation.

Calls to methods from *outside* the method suite may still need to go through the dictionary, however. The situation is familiar from existing object-oriented languages. In Java, for instance, suppose we define two classes A and B and an interface I via the definitions interface I {public int m (int x);} class A implements I { public int m (int x) { ... }; } class B implements I { public int k (int x) { ... }; public int m (int x) { ... }; } In a context where a variable is known only to have type I, a method invocation of m must go through the dictionary: the variable could be an object from the class A (in which case m is the first method in the method table) or from the class B (in which case m is the second method in the method table).

In class-based languages, the only operations that create objects are constructor functions. Thus, when compiling such a language into our calculus, all of the object operations *except* method invocation can be confined to the constructor functions. Constructor functions first call their superclass constructor functions, which return a partially constructed object, and then add or override methods. If the superclass constructor is known—as it is in a language like Java—the dictionaries are known, and so substitutions and compositions of dictionaries can be done at compile time. Even in a language with parameterized classes, one can imagine doing much of the manipulation of dictionaries at *link time* when the base classes of parameterized classes become instantiated.

Any of the optimizations valid for untyped object-oriented languages should apply here as well.

The dynamic semantics does do much more dictionary manipulation (stripping and replacing dictionaries) than one would like to see in an implementation. We have previously described a second-order system whose direct implementation should avoid these, at the cost of more (though individually simpler) language constructs and a more complex type system [27].

4.2. *Related Work*

Our calculi embody solutions to two problems: it provides a characterization of private methods, and supports both subtyping and object extension. Previous work has attempted to address these problems, and it is worth comparing these solutions to ours.

In the context of modeling private components in objects, Fisher and Mitchell [15] give an account of private (as well as protected) methods and fields using abstract types. Abstract types can be used to hide the representations of objects from clients, even though the objects themselves have access to the internal representations. Information about the names of private fields and methods, however, is still exposed. Their account is in some sense more fundamental than ours: our calculus directly supports hiding, and does not attempt to describe it in more basic concepts. Rémy and Vouillon [26] consider a more direct account of private data in classes, but only as inlined constant values. In addition to not matching a standard implementation, their approach does not extend well to mutable fields in the presence of object cloning or functional update of objects. Eiffel [21] has operations for redefining and "undefining" the methods of a class, much like our single renaming operation does in the first-order calculus. We are not aware, however, of any formal accounts that establish the soundness of the Eiffel type system. Bracha and Lindstrom [5] define a coercive operation for hiding components of objects; this appears to behave similarly to our subsumption operation, at least for first-order objects. They formalize this operation within an untyped λ-calculus.

More work has addressed the problems with object extension and subtyping. Fisher and Mitchell [14], for instance, discuss the unsoundness of width subtyping in the presence of object extension. Their solution is to distinguish the types of objects which support either method override and object extension (but no subtyping) from those which support width and depth subtyping but not method override or object extension. Later work has looked at other ways of combining width subtyping with object extension without losing soundness. Liquori [19, 20] gives first- and second-order systems in which the types of extensible objects list the names and types of (a superset of) methods hidden by subsumption; the types must match if the object is extended by a new method with the same name as a hidden method. The idea is related to an old idea: Jategaonkar and Mitchell [17] and Rémy [25] use types

that keep track of which methods must be "absent" from an object. Bono, Bugliesi, Dezani, and Liquori [2, 3, 4] take a different approach: object types contain a conservative approximation of which methods each method invokes via self. A collection of methods can be forgotten via subsumption if no remaining methods might invoke a member of this collection. This is not useful, however, for the purposes of modeling private methods (which exist for the sole purpose of being used by public methods).

It should be noted that even though we allow width-subtyping for objects, the rule for typing object values in a language without object extension can still be more liberal than our Rule 40 [1]. In particular without object extension the type β of the self variable can be known *exactly* within object methods because this cannot be changed by future operations. In contrast, we can only assume that β will be a subtype of the object's current type.

If there is no object extension, we may type the method bodies in an object with the self-type α *equal* to an object type, rather than merely being a subtype of an object type. This permits type-correct objects with "backup" and "restore" methods, where "restore" returns an older copy of the object and has return type α. Since in our system the internal type of the object may change (through object extension), allowing this sort of code in our system would lead to unsoundness.

Finally, in many conventional object-oriented languages, subclassing determines the type hierarchy. Therefore, each method can be associated with the class where it was defined. This allows two co-existing components with the same name to be defined in different classes. The guarantee that related classes will have distinct names guarantees that all references to components can be disambiguated. This does mean that inheritance requires static knowledge about all superclasses, but this is commonly required for implementation efficiency anyway. Java "binary compatibility" allows (among other changes) new methods to be added to an existing class without recompilation of subclasses or other clients; Drossopoulou, Wragg, and Eisenbach [9] give a semantics for this where class names are used to qualify references to object and class components so as to prevent conflicts.

4.3. *Discussion and Future Work*

We have shown that there is a calculus with width subtyping and object extension, one that allows a general notion of strong privacy for fields and methods within classes. Our object calculus does appear to be useful; Fisher and Reppy have been using a variant for the design of an extension to SML with classes and objects [16].

It is not difficult to see how the first-order system embeds in the second-order system. To see how to embed types, a first-order object type $\{\!|l : \tau_l \ ^{l\in I}|\!\}$ can be represented as second-order type $\mathsf{Obj}\,\alpha.\{\!|l : \tau_l \ ^{l\in I}|\!\}$ simply by treating the self type α as a dummy variable. All of the other types embed straightforwardly. To embed the terms, we rewrite method bodies in override and extension so that all occurrences of the self variable s are replaced with $(s@d')$, and we annotate the invocation and override methods with identity dictionaries.

Type-checking for both the first-order and second-order systems is decidable. This fact hinges on a proof that any term in the two systems can be given a minimal typing, i.e., least in the sense of subtyping. What remains open, however, is whether one can build a type inference system that does not force the programmer to write in any types. Type annotations on method bodies are required for minimal typing in the absence of depth subtyping, and depth subtyping is unsound in the presence of method override.

Whether we have chosen the best set of primitives is open to debate. Nevertheless, many extensions should be possible. For instance, it should be possible to add mutable fields and methods and allow imperative update rather than functional update. Variance annotations should also be simple to add to the calculus to support richer forms of subtyping. Finally, it would be interesting to extend the language with bounded polymorphism, which would make the calculus more expressive. We do not anticipate any major difficulties in these directions.

It appears that the systems presented here may be instances of a more general setting in which objects do not carry dictionaries, but rather renamed values are considered values. Thus the first-order object value $\mathsf{obj}\,s.\{\!|i \triangleright e_i : \tau_i \ ^{i\in 1..n}|\!\}_\varphi$ corresponds to the renamed object $(\mathsf{obj}\,s.\{\!|i \triangleright e_i : \tau_i \ ^{i\in 1..n}|\!\})@\varphi$. The dynamic semantics for the first-order system might be

$$(v@\varphi).l \rightsquigarrow v.(\varphi(l))$$
$$(\mathsf{obj}\,s.\{\!|i \triangleright e_i : \tau_i \ ^{i\in I}|\!\}).l \rightsquigarrow [s \mapsto \mathsf{obj}\,s.\{\!|i \triangleright e_i : \tau_i \ ^{i\in I}|\!\}]\,e_l$$

with additional rules such as

$$(v@\varphi)@\varphi' \rightsquigarrow v@(\varphi \circ \varphi')$$

if desired. Then our first-order calculus can be viewed as the special case where we require every object value to be renamed exactly once (by inserting the identity renaming or composing renamings as necessary). It appears possible to view the second-order calculus in a similar way, though with more complicated rules because we must still do substitutions of dictionaries.

Some other, more difficult problems arise, the most important of which is to find a better semantical framework for the calculus. Our proofs of type soundness were purely operational; what would be better is a deeper understanding of the calculus that would make the static semantic rules obvious. A translation of the calculus into a typed λ-calculus might shed some light, or a denotational framework might provide a better setting to evaluate different choices of static rules.

APPENDIX A: STATIC SEMANTICS OF FIRST-ORDER SYSTEM

Well-Formed Contexts $\boxed{\Gamma \vdash \diamond}$

$$\frac{}{\bullet \vdash \diamond} \tag{1}$$

$$\frac{\Gamma \vdash \diamond \qquad x \notin \mathsf{Dom}(\Gamma)}{\Gamma, x{:}\tau \vdash \diamond} \tag{2}$$

Well-Formed Types $\boxed{\Gamma \vdash \tau}$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash b} \tag{3}$$

$$\frac{\Gamma \vdash \tau_1 \qquad \Gamma \vdash \tau_2}{\Gamma \vdash (\tau_1 \to \tau_2)} \tag{4}$$

$$\frac{\Gamma \vdash \diamond \qquad \forall l \in I: \quad \Gamma \vdash \tau_l}{\Gamma \vdash \{\!|\, l : \tau_l \;^{l \in I} \,|\!\}} \tag{5}$$

Width Subtyping $\boxed{\Gamma \vdash \tau_1 \preceq \tau_2}$

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \tau \preceq \tau} \tag{6}$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \{\!|\, l : \tau_l \;^{l \in I \cup J} \,|\!\} \preceq \{\!|\, l : \tau_l \;^{l \in I} \,|\!\}} \tag{7}$$

$$\frac{\Gamma \vdash \tau_1' \preceq \tau_1 \qquad \Gamma \vdash \tau_2 \preceq \tau_2'}{\Gamma \vdash (\tau_1 \to \tau_2) \preceq (\tau_1' \to \tau_2')} \tag{8}$$

Well-Formed Expressions $\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash c : \mathsf{typeof}(c)} \tag{9}$$

$$\frac{\Gamma \vdash \diamond \qquad x \in \mathsf{Dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \tag{10}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash (\lambda x{:}\tau.e) : (\tau \to \tau')} \tag{11}$$

$$\frac{\Gamma \vdash e : (\tau \to \tau') \qquad \Gamma \vdash e' : \tau}{\Gamma \vdash (e\ e') : \tau'} \tag{12}$$

$$\frac{\Gamma \vdash \tau \preceq \tau' \qquad \Gamma \vdash e : \tau}{\Gamma \vdash e : \tau'} \tag{13}$$

$$\frac{\Gamma \vdash e : \{\!|\, l : \tau \,|\!\}}{\Gamma \vdash e.l : \tau} \tag{14}$$

$$\frac{\Gamma \vdash e : \{\!|\, l : \tau_l \ {}^{l \in I} \,|\!\} \qquad \mathsf{Range}(\varphi) \subseteq I}{\Gamma \vdash e@\varphi : \{\!|\, l : \tau_{\varphi(l)} \ {}^{l \in \mathsf{Dom}(\varphi)} \,|\!\}} \tag{15}$$

$$\frac{\begin{array}{c}\mathsf{Range}(\varphi) \subseteq I \\ \forall i \in I : \quad \Gamma, s{:}\{\!|\, m : \tau_m \ {}^{m \in I} \,|\!\} \vdash e_i : \tau_i\end{array}}{\Gamma \vdash \mathsf{obj}\, s.\{\!|\, m \triangleright e_m : \tau_m \ {}^{m \in I} \,|\!\}_\varphi : \{\!|\, l : \tau_{\varphi(l)} \ {}^{l \in \mathsf{Dom}(\varphi)} \,|\!\}} \tag{16}$$

$$\frac{\begin{array}{c}m \in I \qquad s \notin \mathsf{Dom}(\Gamma) \\ \Gamma \vdash e : \{\!|\, l : \tau_l \ {}^{l \in I} \,|\!\} \qquad \Gamma, s{:}\{\!|\, l : \tau_l \ {}^{l \in I} \,|\!\} \vdash e'_m : \tau_m\end{array}}{\Gamma \vdash e \leftarrow m(s) = e'_m : \{\!|\, l : \tau_l \ {}^{l \in I} \,|\!\}} \tag{17}$$

$$\frac{\begin{array}{c}m \notin I \qquad s \notin \mathsf{Dom}(\Gamma) \\ \Gamma \vdash e : \{\!|\, l : \tau_l \ {}^{l \in I} \,|\!\} \qquad \Gamma, s{:}\{\!|\, l : \tau_l \ {}^{l \in I}, m : \tau'_m \,|\!\} \vdash e'_m : \tau'_m\end{array}}{\Gamma \vdash (e \leftarrow\!\!+ m(s) = e'_m : \tau'_m) : \{\!|\, l : \tau_l \ {}^{l \in I}, m : \tau'_m \,|\!\}} \tag{18}$$

APPENDIX B: STATIC SEMANTICS OF SECOND-ORDER SYSTEM

Well-Formed Contexts $\boxed{\Gamma \vdash \diamond}$

$$\frac{}{\bullet \vdash \diamond} \tag{19}$$

$$\frac{\Gamma \vdash \diamond \qquad x \notin \mathsf{Dom}(\Gamma)}{\Gamma, x{:}\tau \vdash \diamond} \tag{20}$$

$$\frac{\Gamma \vdash \tau \qquad \alpha \notin \mathsf{Dom}(\Gamma)}{\Gamma, \alpha \preceq \tau \vdash \diamond} \tag{21}$$

Well-Formed Types $\boxed{\Gamma \vdash \tau}$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash b} \tag{22}$$

$$\frac{\Gamma \vdash \diamond \qquad \alpha \in \mathsf{Dom}(\Gamma)}{\Gamma \vdash \alpha} \tag{23}$$

$$\frac{\Gamma \vdash \tau_1 \qquad \Gamma \vdash \tau_2}{\Gamma \vdash (\tau_1 \to \tau_2)} \tag{24}$$

$$\frac{\Gamma \vdash \tau_1 \qquad \Gamma \vdash \tau_2}{\Gamma \vdash \tau_1 \Rightarrow \tau_2} \tag{25}$$

$$\frac{\begin{array}{c}\Gamma \vdash \diamond \\ \forall l \in I : \quad \Gamma, \alpha \preceq \top \vdash \tau_l\end{array}}{\Gamma \vdash \mathsf{Obj}\, \alpha.\{\!|\, l : \tau_l \ {}^{l \in I} \,|\!\}} \tag{26}$$

Width Subtyping $\boxed{\Gamma \vdash \tau_1 \preceq \tau_2}$

$$\frac{\Gamma \vdash \mathsf{Obj}\,\alpha.\{\!|\, l : \tau_l \;^{l \in I \cup J} \,|\!\}}{\Gamma \vdash \mathsf{Obj}\,\alpha.\{\!|\, l : \tau_l \;^{l \in I \cup J} \,|\!\} \preceq \mathsf{Obj}\,\alpha.\{\!|\, l : \tau_l \;^{l \in I} \,|\!\}} \tag{27}$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \tau \preceq \tau} \tag{28}$$

$$\frac{\Gamma = \Gamma', \alpha \preceq \tau', \Gamma'' \qquad \Gamma \vdash \tau' \preceq \tau}{\Gamma \vdash \alpha \preceq \tau} \tag{29}$$

$$\frac{\Gamma \vdash \tau_1' \preceq \tau_1 \qquad \Gamma \vdash \tau_2 \preceq \tau_2'}{\Gamma \vdash (\tau_1 \to \tau_2) \preceq (\tau_1' \to \tau_2')} \tag{30}$$

$$\frac{\Gamma \vdash \tau_1' \preceq \tau_1 \qquad \Gamma \vdash \tau_2 \preceq \tau_2'}{\Gamma \vdash \tau_1 \Rightarrow \tau_2 \preceq \tau_1' \Rightarrow \tau_2'} \tag{31}$$

Well-Formed Expressions $\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma \vdash e : \tau' \qquad \Gamma \vdash \tau' \preceq \tau}{\Gamma \vdash e : \tau} \tag{32}$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash c : \mathsf{typeof}(c)} \tag{33}$$

$$\frac{\Gamma \vdash \diamond \qquad x \in \mathsf{Dom}(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \tag{34}$$

$$\frac{\Gamma, x{:}\tau \vdash e : \tau'}{\Gamma \vdash (\lambda x{:}\tau.e) : (\tau \to \tau')} \tag{35}$$

$$\frac{\Gamma \vdash e_1 : (\tau_2 \to \tau) \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1\ e_2) : \tau} \tag{36}$$

$$\frac{\begin{array}{c}\Gamma \vdash A \qquad \Gamma \vdash B \qquad \mathsf{Range}(\varphi) \subseteq I \\ B = \mathsf{Obj}\,\alpha.\{\!|\, m : \tau_m \;^{m \in I} \,|\!\} \\ A = \mathsf{Obj}\,\alpha.\{\!|\, l : \tau_{\varphi(l)} \;^{l \in \mathsf{Dom}(\varphi)} \,|\!\}\end{array}}{\Gamma \vdash \varphi : B \Rightarrow A} \tag{37}$$

$$\frac{\Gamma \vdash e : \tau' \qquad \Gamma \vdash v : \tau' \Rightarrow \tau}{\Gamma \vdash e@v : \tau} \tag{38}$$

$$\frac{\Gamma \vdash e : \tau' \qquad \Gamma \vdash v : \tau' \Rightarrow \mathsf{Obj}\,\alpha.\{\!|\, l : \tau \,|\!\}}{\Gamma \vdash e._v l : [\alpha \mapsto \tau']\,\tau} \tag{39}$$

$$\frac{\begin{array}{c}\forall m \in I : \quad \beta \notin \mathsf{FV}(\tau_m) \\ B = \mathsf{Obj}\,\alpha.\{\!|\, m : \tau_m \;^{m \in I} \,|\!\} \\ A = \mathsf{Obj}\,\alpha.\{\!|\, l : \tau_{\varphi(l)} \;^{l \in \mathsf{Dom}(\varphi)} \,|\!\} \\ \Gamma \vdash \varphi : B \Rightarrow A \\ \forall m \in I : \quad \Gamma, \beta \preceq B, \alpha \preceq \top, d{:}\beta \Rightarrow \alpha, s{:}\beta \vdash e_m : \tau_m\end{array}}{\Gamma \vdash \mathsf{obj}(\alpha, \beta, s, d).\{\!|\, m \triangleright e_m : \tau_m \;^{m \in I} \,|\!\}_\varphi : A} \tag{40}$$

$$\frac{\begin{array}{c}\beta \notin \mathsf{FV}(\tau_m) \\ \Gamma \vdash e_1 : \tau \qquad \Gamma \vdash v : \tau \Rightarrow \mathsf{Obj}\,\alpha.\{\!|\, m : \tau_m \,|\!\} \\ \Gamma, \beta \preceq \top, \alpha \preceq \top, d'{:}\beta \Rightarrow \tau, d{:}\beta \Rightarrow \alpha, s{:}\beta \vdash e_2 : \tau_m\end{array}}{\Gamma \vdash e_1 \leftarrow_v m(\alpha, \beta, s, d, d') = e_2 : \tau} \tag{41}$$

$$\frac{\begin{array}{c}\beta \notin \mathsf{FV}(\tau) \qquad m \notin I \\ \Gamma \vdash e_1 : \mathsf{Obj}\,\alpha.\{\!|\, l : \tau_l \;^{l\in I} \,|\!\} \\ A = \mathsf{Obj}\,\alpha.\{\!|\, l : \tau_l \;^{l\in I}, m : \tau \,|\!\} \\ \Gamma, \beta \preceq \top, \alpha \preceq \top, d' : \beta \Rightarrow A, d : \beta \Rightarrow \alpha, s : \beta \vdash e_2 : \tau\end{array}}{\Gamma \vdash (e_1 \leftarrow\!\!+ m(\alpha, \beta, s, d, d') = e_2 : \tau) : A} \tag{42}$$

APPENDIX C: CORRECTNESS OF FIRST-ORDER SYSTEM

We first give a series of lemmas. The proofs are omitted because they are largely similar to (but simpler than) the corresponding lemmas in the second-order system, which appear in Appendix D. We use the notation $\mathcal{J}$ for the right-hand side of a judgement in the system.

LEMMA C.1 (Well-Formedness). *If* $\Gamma, \Gamma' \vdash \diamond$ *then* $\Gamma \vdash \diamond$.

LEMMA C.2. *If* $\Gamma \vdash \mathcal{J}$ *then* $\Gamma \vdash \diamond$.

LEMMA C.3. *If* $\Gamma, \Gamma' \vdash \diamond$ *then* $\mathsf{Dom}(\Gamma) \cap \mathsf{Dom}(\Gamma') = \emptyset$.

LEMMA C.4 (Context Weakening). *If* $\Gamma, \Gamma'' \vdash \mathcal{J}$ *and* $\Gamma, \Gamma', \Gamma'' \vdash \diamond$ *then* $\Gamma, \Gamma', \Gamma'' \vdash \mathcal{J}$.

LEMMA C.5 (Bound Weakening). *If* $\Gamma_1, x : \tau_2, \Gamma_2 \vdash \mathcal{J}$ *and* $\Gamma_1 \vdash \tau_1 \preceq \tau_2$ *then* $\Gamma_1, x : \tau_1, \Gamma_2 \vdash \mathcal{J}$.

LEMMA C.6 (Transitivity). *If* $\Gamma \vdash \tau \preceq \tau'$ *and* $\Gamma \vdash \tau' \preceq \tau''$ *then* $\Gamma \vdash \tau \preceq \tau''$.

LEMMA C.7 (Subtyping Inversion).

i. *If* $\Gamma \vdash (\tau_1' \to \tau_2) \preceq (\tau_1 \to \tau_2')$ *then* $\Gamma \vdash \tau_i \preceq \tau_i'$.
ii. *If* $\Gamma \vdash \{\!|\, l : \tau_l \;^{l\in L} \,|\!\} \preceq \{\!|\, l : \tau_l' \;^{l\in L'} \,|\!\}$ *then* $L' \subseteq L$ *and* $\tau_l' = \tau_l$ *for all* $l \in L'$.

LEMMA C.8 (Value Substitution). *If* $\Gamma, x{:}\tau \vdash e' : \tau'$ *and* $\Gamma, \Gamma' \vdash e : \tau$, *then* $\Gamma, \Gamma' \vdash [x \mapsto e]\, e' : \tau'$.

LEMMA C.9 (Decomposition and Replacement). *If* $\Gamma \vdash E[e] : \tau$ *then* $\Gamma \vdash e : \tau'$ *for some type* τ'. *Furthermore, if* $\Gamma \vdash e' : \tau'$ *then* $\Gamma \vdash E[e'] : \tau$.

THEOREM C.10 (Subject Reduction). *If* $\Gamma \vdash e : \sigma$ *and* $e \rightsquigarrow e'$ *then* $\Gamma \vdash e' : \sigma$.

Proof. By cases depending on the operational rule used. Because of the transitivity and symmetry of subtyping, without loss of generality we may assume that in the proof of $\Gamma \vdash e : \sigma$, uses of the Subsumption Rule 13 alternate with uses of the remaining typing rules, and that the proof does not end with a use of subsumption. By Lemma C.9 we need only consider the local reduction steps.

1. The rule used is

$$(\lambda x{:}\tau.e)\; v \rightsquigarrow [x \mapsto v]\, e.$$

Then the proof of $\Gamma \vdash (\lambda x{:}\tau.e)\; v : \sigma$ must contain sub-proofs of the following judgments:

$$\begin{array}{l}\Gamma, x{:}\tau \vdash e : \tau_1 \\ \Gamma \vdash (\tau \to \tau_1) \preceq (\tau_2 \to \sigma) \\ \Gamma \vdash v : \tau_2\end{array}$$

By Lemma C.7, we have $\Gamma \vdash \tau_2 \preceq \tau$ and $\Gamma \vdash \tau_1 \preceq \sigma$. Therefore $\Gamma \vdash v : \tau$ by subsumption, $\Gamma \vdash [x \mapsto v]\, e : \tau_1$ by Lemma C.8, and the desired result follows by subsumption.

2. The rule used is

$$\begin{array}{r}\mathsf{obj}\, s.\{\!|\, m \triangleright e_m : \tau_m \;^{m\in I} \,|\!\}_\varphi @ \varphi' \rightsquigarrow \\ \mathsf{obj}\, s.\{\!|\, m \triangleright e_m : \tau_m \;^{m\in I} \,|\!\}_{\varphi\circ\varphi'}\end{array}$$

The typing proof of the hypothesis must include the following judgments:

$$\begin{array}{l}\forall i \in I: \quad \Gamma, s{:}\{\!|m : \tau_m \ ^{m\in I}|\!\} \vdash e_i : \tau_i\\ \mathsf{Range}(\varphi) \subseteq I\\ \Gamma \vdash \{\!|m : \tau_{\varphi(m)} \ ^{m\in \mathsf{Dom}(\varphi)}|\!\} \preceq \{\!|m : \tau'_m \ ^{m\in L}|\!\}\\ \mathsf{Range}(\varphi') \subseteq L\end{array}$$

where

$$\sigma = \{\!|m : \tau'_{\varphi'(m)} \ ^{m\in \mathsf{Dom}(\varphi')}|\!\}.$$

By Lemma C.7, $L \subseteq \mathsf{Dom}(\varphi)$ and $\tau'_m = \tau_{\varphi(m)}$ for all $m \in L$. Then $\mathsf{Range}(\varphi \circ \varphi') \subseteq \mathsf{Range}(\varphi) \subseteq I$ and $\mathsf{Dom}(\varphi \circ \varphi') = \mathsf{Dom}(\varphi')$. Therefore, $\Gamma \vdash \mathsf{obj}\, s \cdot \{\!|m \triangleright e_m : \tau_m \ ^{m\in I}|\!\}_{\varphi\circ\varphi'} : \{\!|m : \tau_{\varphi(\varphi'(m))} \ ^{m\in \mathsf{Dom}(\varphi\circ\varphi')}|\!\}$ and this type is equal to σ.

3. The rule used is

$$\begin{array}{l}(\mathsf{obj}\, s.\{\!|m \triangleright e_m : \tau_m \ ^{m\in I}|\!\}_\varphi).l \rightsquigarrow\\ \quad [s \mapsto \mathsf{obj}\, s.\{\!|m \triangleright e_m : \tau_m \ ^{m\in I}|\!\}_{id(I)}]\, e_{\varphi(l)}\end{array}$$

The typing proof of the hypothesis must include the following judgments:

$$\begin{array}{l}\forall i \in I: \quad \Gamma, s{:}\{\!|m : \tau_m \ ^{m\in I}|\!\} \vdash e_i : \tau_i\\ \mathsf{Range}(\varphi) \subseteq I\\ \Gamma \vdash \{\!|m : \tau_{\varphi(m)} \ ^{m\in \mathsf{Dom}(\varphi)}|\!\} \preceq \{\!|l : \sigma|\!\}\end{array}$$

By Lemma C.7, we have $l \in \mathsf{Dom}(\varphi)$ and $\sigma = \tau_{\varphi(l)}$. Further, $\Gamma \vdash \mathsf{obj}\, s.\{\!|m \triangleright e_m : \tau_m \ ^{m\in I}|\!\}_{id(I)} : \{\!|m : \tau_m \ ^{m\in I}|\!\}$. Finally, by Lemma C.8, it follows that $\Gamma \vdash [s \mapsto \mathsf{obj}\, s.\{\!|m \triangleright e_m : \tau_m \ ^{m\in I}|\!\}_{id(I)}]\, e_{\varphi(l)} : \tau_{\varphi(l)}$ as desired.

4. The rule used is

$$\begin{array}{l}(\mathsf{obj}\, s.\{\!|m \triangleright e_m : \tau_m \ ^{m\in I}|\!\}_\varphi) \leftarrow l(s) = e \rightsquigarrow\\ \quad \mathsf{obj}\, s.\{\!|\, m \triangleright e_m : \tau_m \ ^{m\in I\setminus\varphi(l)},\\ \qquad \varphi(l) \triangleright [s \mapsto s@\varphi]\, e : \tau_{\varphi(l)}|\!\}_\varphi\end{array}$$

The typing proof of the hypothesis must include the following judgments:

$$\begin{array}{l}\forall i \in I: \quad \Gamma, s{:}\{\!|m : \tau_m \ ^{m\in I}|\!\} \vdash e_i : \tau_i\\ \mathsf{Range}(\varphi) \subseteq I\\ \Gamma \vdash \{\!|m : \tau_{\varphi(m)} \ ^{m\in \mathsf{Dom}(\varphi)}|\!\} \preceq \{\!|m : \tau'_m \ ^{m\in L}|\!\}\\ \Gamma, s{:}\{\!|m : \tau'_m \ ^{m\in L}|\!\} \vdash e : \tau'_l\\ l \in L\end{array}$$

where $\sigma = \{\!|m : \tau'_m \ ^{m\in L}|\!\}$. By Lemma C.7, $L \subseteq \mathsf{Dom}(\varphi)$ and $\tau'_m = \tau_{\varphi(m)}$ for all $m \in L$. Then

$$\Gamma, s{:}\{\!|m : \tau_m \ ^{m\in I}|\!\} \vdash s@\varphi : \{\!|m : \tau_{\varphi(m)} \ ^{m\in \mathsf{Dom}(\varphi)}|\!\}$$

which is a subtype of $\{\!|m : \tau_{\varphi(m)} \ ^{m\in L}|\!\} = \{\!|m : \tau'_m \ ^{m\in L}|\!\}$. Thus, by subsumption,

$$\Gamma, s{:}\{\!|m : \tau_m \ ^{m\in I}|\!\} \vdash s@\varphi : \{\!|m : \tau'_m \ ^{m\in L}|\!\}$$

By Lemma C.8, therefore,

$$\Gamma, s{:}\{\!|m : \tau_m \ ^{m\in I}|\!\} \vdash [s \mapsto s@\varphi]\, e : \tau'_l$$

and so we have all the pieces to construct the typing proof for the reduced term.

5. The rule used is

$$\begin{array}{l}(\mathsf{obj}\, s.\{\!|m \triangleright e_m : \tau_m \ ^{m\in I}|\!\}_\varphi) \leftarrow\!\!+ l(s) = e : \tau \rightsquigarrow \\ \quad \mathsf{obj}\, s.\{\!|m \triangleright e_m : \tau_m \ ^{m\in I}, l' \triangleright [s \mapsto s@\varphi']\, e : \tau|\!\}_{\varphi'}\end{array}$$

where $\varphi' = \varphi[l \mapsto l']$. The typing proof of the hypothesis must include the following judgments:

$$\begin{array}{l}\forall i \in I : \quad \Gamma, s{:}\{\!|m : \tau_m \ ^{m\in I}|\!\} \vdash e_i : \tau_i \\ \mathsf{Range}(\varphi) \subseteq I \\ \Gamma \vdash \{\!|m : \tau_{\varphi(m)} \ ^{m\in \mathsf{Dom}(\varphi)}|\!\} \preceq \{\!|m : \tau'_m \ ^{m\in L}|\!\} \\ \Gamma, s{:}\{\!|m : \tau'_m \ ^{m\in L}|\!\} \vdash e : \tau \\ l \notin L\end{array}$$

where $\sigma = \{\!|m : \tau'_m \ ^{m\in L}|\!\}$. Again Lemma C.7 gives us $L \subseteq \mathsf{Dom}(\varphi)$ and $\tau'_m = \tau_{\varphi(m)}$ for all $m \in L$. By Lemma C.5 we can show that

$$\forall i \in I : \quad \Gamma, s{:}\{\!|m : \tau_m \ ^{m\in I}, l' : \tau|\!\} \vdash e_i : \tau_i$$

In a similar way, the conclusion then follows the same outline as the previous case, with φ' in place of φ.

This completes the case analysis and hence the proof. ■

LEMMA C.11 (Canonical Forms). *If* $\vdash v : (\tau \to \tau')$ *then* v *is a lambda expression. If* $\vdash v : \{\!|m : \tau_m \ ^{m\in I}|\!\}$ *then* v *is an object value.*

THEOREM C.12 (Progress). *If* $\vdash e : \tau$ *then either* e *is a value or else* $e \rightsquigarrow e'$.

APPENDIX D: CORRECTNESS OF SECOND-ORDER SYSTEM

The proof requires a number of simple lemmas. We use the notation $\mathcal{J}$ for the right-hand side of a judgement in the system.

LEMMA D.12 (Well-Formedness). *If* $\Gamma, \Gamma' \vdash \diamond$ *then* $\Gamma \vdash \diamond$.

Proof. By Rules 20 and 21, we can drop the last entry in a well-formed context and still have a well-formed context. By a simple inductive argument, we can therefore drop an arbitrary end portion of a context and retain well-formedness. ■

LEMMA D.13. *If* $\Gamma \vdash \mathcal{J}$ *then* $\Gamma \vdash \diamond$.

Proof. By induction on the proof of the premise. ■

LEMMA D.14. *If* $\Gamma, \Gamma' \vdash \diamond$ *then* $\mathsf{Dom}(\Gamma) \cap \mathsf{Dom}(\Gamma') = \emptyset$.

Proof. By induction on the proof of $\Gamma, \Gamma' \vdash \diamond$. ■

LEMMA D.15 (Transitivity). *If* $\Gamma \vdash \tau \preceq \tau'$ *and* $\Gamma \vdash \tau' \preceq \tau''$ *then* $\Gamma \vdash \tau \preceq \tau''$.

Proof. By cases on the final rules of the two derivations.

- 27& 27. By transitivity of set inclusion.
- 28&* or *&28. Trivial.
- 29&*. Then $\Gamma \vdash \alpha \preceq \tau$, where $\Gamma = \Gamma', \alpha \preceq \sigma, \Gamma''$ and $\Gamma \vdash \sigma \preceq \tau'$. By the inductive hypothesis, $\Gamma \vdash \sigma \preceq \tau''$. Thus, by Rule 29, $\Gamma \vdash \alpha \preceq \tau''$.
- 30&30, 31&31. These cases follow directly from the inductive hypothesis.

This completes the case analysis and hence the proof. ■

LEMMA D.16 (Subtyping Inversion).

i. *If* $\Gamma \vdash (\tau_1' \to \tau_2) \preceq (\tau_1 \to \tau_2')$ *then* $\Gamma \vdash \tau_i \preceq \tau_i'$.

ii. *If* $\Gamma \vdash \tau_1' \Rightarrow \tau_2 \preceq \tau_1 \Rightarrow \tau_2'$ *then* $\Gamma \vdash \tau_i \preceq \tau_i'$.

Proof. Follows from the fact that the proofs of the hypotheses must end with Rule 28 or Rule 30, and with Rule 28 or Rule 31, respectively. ■

LEMMA D.17 (Context Weakening). *If* $\Gamma, \Gamma'' \vdash \mathcal{J}$ *and* $\Gamma, \Gamma', \Gamma'' \vdash \diamond$ *then* $\Gamma, \Gamma', \Gamma'' \vdash \mathcal{J}$.

Proof. By induction on proof of first premise, and cases on the last rule used. We give a few representative cases and leave the others to the reader.

- 29. Then $\Gamma, \Gamma'' \vdash \alpha \preceq \tau$, where $\Gamma, \Gamma'' = \Gamma_1, \alpha \preceq \tau', \Gamma_2$ and $\Gamma, \Gamma'' \vdash \tau' \preceq \tau$. By the inductive hypothesis, $\Gamma, \Gamma', \Gamma'' \vdash \tau' \preceq \tau$. Note also that $\Gamma, \Gamma', \Gamma'' = \Gamma_1', \alpha \preceq \tau', \Gamma_2'$. Thus, by Rule 29,

$$\Gamma, \Gamma', \Gamma'' \vdash \alpha \preceq \tau$$

as desired.

- 34. Then $\Gamma, \Gamma'' \vdash x : (\Gamma, \Gamma'')(x)$, where $\Gamma, \Gamma'' \vdash \diamond$. By Lemma D.14, we have $\mathsf{Dom}(\Gamma') \cap (\mathsf{Dom}(\Gamma) \cup \mathsf{Dom}(\Gamma'')) = \emptyset$. Therefore $x \notin \mathsf{Dom}(\Gamma')$, so $(\Gamma, \Gamma'')(x) = (\Gamma, \Gamma', \Gamma'')(x)$. The conclusion follows from the second premise.

- 35. By α-conversion, we may assume $x \notin \mathsf{Dom}(\Gamma')$. The conclusion follows from the inductive hypothesis.

This completes the case analysis and hence the proof. ■

LEMMA D.18 (Bound Weakening). *If* $\Gamma_1, \beta \preceq \tau_2, \Gamma_2 \vdash \mathcal{J}$ *and* $\Gamma_1 \vdash \tau_1 \preceq \tau_2$ *then* $\Gamma_1, \beta \preceq \tau_1, \Gamma_2 \vdash \mathcal{J}$.

Proof. By induction on proof of first premise, and cases on the last rule used. The only difficult case is when the last rule used is Rule 29; we give this case and leave the others to the reader. We know that $\Gamma_1, \beta \preceq \tau_2, \Gamma_2 \vdash \alpha \preceq \tau$, where $\Gamma_1, \beta \preceq \tau_2, \Gamma_2 = \Gamma, \alpha \preceq \tau', \Gamma'$ and $\Gamma_1, \beta \preceq \tau_2, \Gamma_2 \vdash \tau' \preceq \tau$. There are two cases:

- $\beta = \alpha$. Then $\tau_2 = \tau'$. By Context Weakening applied to the second hypothesis we have

$$\Gamma_1, \beta \preceq \tau_1, \Gamma_2 \vdash \tau_1 \preceq \tau_2 = \tau'.$$

By the inductive hypothesis,

$$\Gamma_1, \beta \preceq \tau_1, \Gamma_2 \vdash \tau' \preceq \tau$$

so by Transitivity, $\Gamma_1, \beta \preceq \tau_1, \Gamma_2 \vdash \tau_1 \preceq \tau$. Thus, by Rule 29,

$$\Gamma_1, \beta \preceq \tau_1, \Gamma_2 \vdash \alpha \preceq \tau$$

as desired.

- $\beta \neq \alpha$. Then the binding $(\alpha \preceq \tau')$ appears in either Γ_1 or Γ_2. By the inductive hypothesis,

$$\Gamma_1, \beta \preceq \tau_1, \Gamma_2 \vdash \tau' \preceq \tau$$

and so by Rule 29,

$$\Gamma_1, \beta \preceq \tau_1, \Gamma_2 \vdash \alpha \preceq \tau$$

as desired.

This completes the case analysis and hence the proof. ■

LEMMA D.19 (Covariant Substitution). *If α appears covariantly in τ and $\Gamma \vdash \sigma \preceq \sigma'$, then $\Gamma \vdash [\alpha \mapsto \sigma]\,\tau \preceq [\alpha \mapsto \sigma']\,\tau$.*

Proof. We prove the statement above and the following statement

If α appears contravariantly in τ and $\Gamma \vdash \sigma \preceq \sigma'$, then $\Gamma \vdash [\alpha \mapsto \sigma']\,\tau \preceq [\alpha \mapsto \sigma]\,\tau$.

The proof goes by simultaneous induction on τ.

- To see the first statement, if α does not appear free in τ, then the conclusion follows from Rule 28. If $\tau = \alpha$ then the conclusion is exactly the second assumption. Otherwise $\tau = (\tau_1 \to \tau_2)$ or $\tau_1 \Rightarrow \tau_2$ where α appears contravariantly in τ_1 and covariantly in τ_2. In either case the inductive hypothesis gives us $\Gamma \vdash [\alpha \mapsto \sigma']\,\tau_1 \preceq [\alpha \mapsto \sigma]\,\tau_1$ and $\Gamma \vdash [\alpha \mapsto \sigma]\,\tau_2 \preceq [\alpha \mapsto \sigma']\,\tau_2$; the conclusion follows by rule 30 or 31 respectively.
- To see the second statement, if α does not appear free in τ, then the conclusion follows from Rule 28. Otherwise $\tau = (\tau_1 \to \tau_2)$ or $\tau_1 \Rightarrow \tau_2$ where α appears covariantly in τ_1 and contravariantly in τ_2. In either case the inductive hypothesis gives us $\Gamma \vdash [\alpha \mapsto \sigma]\,\tau_1 \preceq [\alpha \mapsto \sigma']\,\tau_1$ and $\Gamma \vdash [\alpha \mapsto \sigma']\,\tau_2 \preceq [\alpha \mapsto \sigma]\,\tau_2$; the conclusion follows by rule 30 or 31 respectively.

This completes the proof. ■

LEMMA D.20 (Type Substitution). *If $\Gamma, \beta \preceq \tau', \Gamma' \vdash \mathcal{J}$, and $\Gamma \vdash \tau \preceq \tau'$ and $\beta \notin \mathsf{FV}(\tau) \cup \mathsf{FV}(\tau')$ then $\Gamma, [\beta \mapsto \tau]\,\Gamma' \vdash [\beta \mapsto \tau]\,\mathcal{J}$.*

Proof. By induction on proof of first premise, and cases on the rule used for the conclusion. We give two of the more difficult cases and leave the others to the reader.

- 21. There are two cases.

  – $\Gamma, \beta \preceq \tau', \Gamma', \alpha \preceq \tau'' \vdash \diamond$, where $\Gamma, \beta \preceq \tau', \Gamma' \vdash \diamond$ and $\alpha \notin \mathsf{Dom}(\Gamma, \beta \preceq \tau', \Gamma')$. By the inductive hypothesis, $\Gamma, [\beta \mapsto \tau]\,\Gamma' \vdash \diamond$. Thus, since $\alpha \notin \mathsf{Dom}(\Gamma, [\beta \mapsto \tau]\,\Gamma')$, it follows by Rule 21 that

$$\Gamma, [\beta \mapsto \tau]\,\Gamma', \alpha \preceq [\beta \mapsto \tau]\,\tau'' \vdash \diamond$$

as desired.

  – $\Gamma, \beta \preceq \tau', \Gamma' \vdash \diamond$, where $\Gamma \vdash \diamond$ and $\alpha \notin \mathsf{Dom}(\Gamma)$ and $\beta = \alpha$. Then $\Gamma' = \bullet$. It follows that $[\beta \mapsto \tau]\,\Gamma' = \Gamma' = \bullet$, so

$$\Gamma, [\beta \mapsto \tau]\,\Gamma' \vdash \diamond$$

as desired.

- 29. Then $\Gamma, \beta \preceq \tau', \Gamma' \vdash \alpha \preceq \tau_2$, where $(\Gamma, \beta \preceq \tau', \Gamma') = (\Gamma_1, \alpha \preceq \tau_1, \Gamma_2)$ and $\Gamma, \beta \preceq \tau', \Gamma' \vdash \tau_1 \preceq \tau_2$. By the inductive hypothesis,

$$\Gamma, [\beta \mapsto \tau]\,\Gamma' \vdash [\beta \mapsto \tau]\,\tau_1 \preceq [\beta \mapsto \tau]\,\tau_2.$$

  There are two cases. If $\beta = \alpha$, then $\tau' = \tau_1$. By Context Weakening $\Gamma, [\beta \mapsto \tau]\,\Gamma' \vdash \tau \preceq \tau'$. Since $\beta \notin \mathsf{FV}(\tau) \cup \mathsf{FV}(\tau')$, we know that $[\beta \mapsto \tau]\,\tau_1 = \tau_1$. Thus,

$$\Gamma, [\beta \mapsto \tau]\,\Gamma' \vdash [\beta \mapsto \tau]\,\alpha \preceq [\beta \mapsto \tau]\,\tau_1$$

and so by Transitivity,

$$\Gamma, [\beta \mapsto \tau]\,\Gamma' \vdash [\beta \mapsto \tau]\,\alpha \preceq [\beta \mapsto \tau]\,\tau_2$$

as desired.

  For the other case, when $\beta \neq \alpha$, note that $[\beta \mapsto \tau]\,\alpha = \alpha$. Also, the binding $(\alpha \preceq \tau_1)$ appears either in Γ or in Γ'. If it appears in Γ, then $\beta \notin \mathsf{FV}(\tau_1)$. Thus, $[\beta \mapsto \tau]\,\tau_1 = \tau_1$, so therefore

$$\Gamma, [\beta \mapsto \tau]\,\Gamma' \vdash \tau_1 \preceq [\beta \mapsto \tau]\,\tau_2.$$

Thus, by Rule 29, it follows that

$$\Gamma, [\beta \mapsto \tau]\,\Gamma' \vdash [\beta \mapsto \tau]\,\alpha \preceq [\beta \mapsto \tau]\,\tau_2.$$

If, on the other hand, the binding $(\alpha \preceq \tau_1)$ appears in Γ', then the binding $(\alpha \preceq [\beta \mapsto \tau]\,\tau_1)$ appears in $[\beta \mapsto \tau]\,\Gamma'$. Thus, by Rule 29, it follows that

$$\Gamma, [\beta \mapsto \tau]\,\Gamma' \vdash [\beta \mapsto \tau]\,\alpha \preceq [\beta \mapsto \tau]\,\tau_2$$

as desired.

This completes the case analysis and hence the proof. ■

LEMMA D.21 (Value Substitution). *If* $\Gamma, x{:}\tau, \Gamma' \vdash e' : \tau'$ *and* $\Gamma, \Gamma' \vdash e : \tau$ *then* $\Gamma, \Gamma' \vdash [x \mapsto e]\,e' : \tau'$.

Proof. By induction on proof of first premise, and cases on the rule used for the conclusion. We give a few representative cases and leave the others to the reader.

- 34. Then $\Gamma, x{:}\tau, \Gamma' \vdash y : \tau'$, where $\Gamma, x{:}\tau, \Gamma' \vdash \diamond$ and $(\Gamma, x{:}\tau, \Gamma')(y) = \tau'$. There are two cases. If $y \neq x$, then a binding $(y{:}\tau')$ appears in Γ, Γ'. Thus, since $[x \mapsto e]\,y = y$

$$\Gamma, \Gamma' \vdash [x \mapsto e]\,y : \tau'.$$

If $y = x$, then $\tau = \tau'$ and $[x \mapsto e]\,y = e$. Thus, by the second hypothesis,

$$\Gamma, \Gamma' \vdash [x \mapsto e]\,y : \tau'$$

as desired.

- 35. Then $\Gamma, x{:}\tau, \Gamma' \vdash \lambda y{:}\tau_1.e_1 : (\tau_1 \to \tau_2)$, where $\Gamma, x{:}\tau, \Gamma', y{:}\tau_1 \vdash e_1 : \tau_2$ and $y \notin \mathsf{dom}(\Gamma, x{:}\tau, \Gamma')$. Thus, we know that $x \neq y$. By the inductive hypothesis,

$$\Gamma, \Gamma', y{:}\tau_1 \vdash [x \mapsto e]\,e_1 : \tau_2$$

so by Rule 35,

$$\Gamma, \Gamma' \vdash [x \mapsto e]\,(\lambda y{:}\tau_1.e_1) : (\tau_1 \to \tau_2)$$

as desired.

This completes the case analysis and hence the proof. ■

LEMMA D.22 (Decomposition and Replacement). *If* $\Gamma \vdash E[e] : \tau$ *then* $\vdash e : \tau'$ *for some type* τ'. *Furthermore, if* $\vdash e' : \tau'$ *then* $\vdash E[e'] : \tau$.

Proof. By induction on the proof of $\Gamma \vdash E[e] : \tau$. ■

THEOREM D.23 (Subject Reduction). *If* $\Gamma \vdash e : \tau$ *and* $e \rightsquigarrow e'$ *then* $\Gamma \vdash e' : \tau$.

Proof. By cases depending on the operational rule used. Because of the transitivity and symmetry of subtyping, without loss of generality we may assume that in the proof of $\Gamma \vdash e : \tau$, uses of the Subsumption Rule 32 alternate with uses of the remaining typing rules, and that the proof does not end with a use of subsumption. By Lemma D.22 we need only consider the local reduction steps.

1. The last rule used is

$$e = (\lambda x{:}\tau_0.e_1)\; v \rightsquigarrow [x \mapsto v]\,e_1 = e'.$$

This case is unchanged from the proof of Theorem C.10.

2. The rule used is

$$\begin{aligned}&(\mathsf{obj}(\alpha, \beta, s, d).\{\!|m \triangleright e_m : \tau_m \ ^{m\in I}|\!\}_\varphi)@\varphi' \\ &\quad \rightsquigarrow \mathsf{obj}(\alpha, \beta, s, d).\{\!|m \triangleright e_m : \tau_m \ ^{m\in I}|\!\}_{\varphi\circ\varphi'}\end{aligned}$$

The proof of the first assumption must end in a use of rule 38. By inspection, the proof must involve the Object Rule 40 and the Dictionary Rule 37 and subsumption, so there must exist derivations:

$$\begin{aligned}&\Gamma \vdash \mathsf{obj}(\alpha, \beta, s, d).\{\!|m \triangleright e_m : \tau_m \ ^{m\in I}|\!\}_\varphi : A \\ &\Gamma, \beta \preceq B, \alpha \preceq \top, d : \beta \Rightarrow \alpha, s{:}\beta \vdash e_m : \tau_m, \ (\forall m \in I) \\ &\Gamma \vdash A \preceq \tau' \\ &\Gamma \vdash \varphi' : \tau'' \Rightarrow \tau''' \\ &\Gamma \vdash \tau'' \Rightarrow \tau''' \preceq \tau' \Rightarrow \tau\end{aligned}$$

where

$$\begin{aligned}A &= \mathsf{Obj}\,\alpha.\{\!|l : \tau_{\varphi(l)} \ ^{l\in\mathsf{Dom}(\varphi)}|\!\} \\ B &= \mathsf{Obj}\,\alpha.\{\!|m : \tau_m \ ^{m\in I}|\!\}\end{aligned}$$

By Subtyping Inversion, we know that $\Gamma \vdash \tau' \preceq \tau''$, and hence $\tau'' = \mathsf{Obj}\,\alpha.\{\!|l : \tau_{\varphi(l)} \ ^{l\in J}|\!\}$ where $J \subseteq \mathsf{Dom}(\varphi)$. By the Dictionary Rule 37, we know that

$$\tau''' = \mathsf{Obj}\,\alpha.\{\!|n : \tau_{\varphi(\varphi'(n))} \ ^{n\in K}|\!\}$$

for some $K \subseteq \mathsf{Dom}(\varphi')$. Since $\mathsf{Dom}(\varphi') = \mathsf{Dom}(\varphi \circ \varphi')$, by the Object Rule 40 and Subsumption,

$$\Gamma \vdash \mathsf{obj}(\alpha, \beta, s, d).\{\!|m \triangleright e_m : \tau_m \ ^{m\in I}|\!\}_{\varphi\circ\varphi'} : \tau'''$$

By Subtyping Inversion, $\Gamma \vdash \tau''' \preceq \tau$, so by Subsumption,

$$\Gamma \vdash \mathsf{obj}(\alpha, \beta, s, d).\{\!|m \triangleright e_m : \tau_m \ ^{m\in I}|\!\}_{\varphi\circ\varphi'} : \tau$$

as desired.

3. The rule used is

$$\begin{aligned}&(\mathsf{obj}(\alpha, \beta, s, d).\{\!|m \triangleright e_m : \tau_m \ ^{m\in I}|\!\}_\varphi)\cdot_{\varphi'} l \\ &\quad \rightsquigarrow [d \mapsto \varphi]\,[s \mapsto \mathit{self}]\,[\alpha \mapsto A]\,[\beta \mapsto B]\,e_{\varphi(\varphi'(l))}\end{aligned}$$

where

$$\begin{aligned}A &= \mathsf{Obj}\,\alpha.\{\!|l : \tau_{\varphi(l)} \ ^{l\in\mathsf{Dom}(\varphi)}|\!\} \\ B &= \mathsf{Obj}\,\alpha.\{\!|m : \tau_m \ ^{m\in I}|\!\} \\ \mathit{self} &= \mathsf{obj}(\alpha, \beta, s, d).\{\!|m \triangleright e_m : \tau_m \ ^{m\in I}|\!\}_{id(I)}\end{aligned}$$

By hypothesis,

$$\Gamma \vdash (\mathsf{obj}(\alpha, \beta, s, d).\{\!|m \triangleright e_m : \tau_m \ ^{m\in I}|\!\}_\varphi).{}_{\varphi'} l : \tau.$$

The derivation for the first premise must end with a use of rule 39. By inspection of the rule, there exist derivations for

$$\begin{aligned}&\Gamma \vdash \mathsf{obj}(\alpha, \beta, s, d).\{\!|m \triangleright e_m : \tau_m \ ^{m\in I}|\!\}_\varphi : \tau_1' \\ &\Gamma \vdash \varphi' : \tau_1' \Rightarrow \mathsf{Obj}\,\alpha.\{\!|l : \tau_2'|\!\}\end{aligned}$$

where $\tau = [\alpha \mapsto \tau_1']\,\tau_2'$. Hence $\Gamma \vdash A \preceq \tau_1'$ and there is a derivation

$$\Gamma \vdash \mathsf{obj}(\alpha, \beta, s, d).\{\!|m \triangleright \tau_m : e_m \;^{m\in I}|\!\}_\varphi : A$$

whose last rule is the Object Rule 40. Thus, for all $m \in I$, there is a derivation

$$\Gamma, \beta \preceq B, \alpha \preceq \top, d : \beta \Rightarrow \alpha, s{:}\beta \vdash e_m : \tau_m$$

Now, $\Gamma \vdash \varphi' : \tau_1' \Rightarrow \mathsf{Obj}\,\alpha.\{\!|l : \tau_2'|\!\}$ implies that $\tau_{\varphi(\varphi'(l))} = \tau_2'$. Let $\tau'' = [\alpha \mapsto A]\,[\beta \mapsto B]\,\tau_{\varphi(\varphi'(l))}$. By the Type Substitution Lemma,

$$\Gamma, d : B \Rightarrow A, s{:}B \vdash [\alpha \mapsto A]\,[\beta \mapsto B]\,e_{\varphi(\varphi'(l))} : \tau''$$

Note that by the Object Rule 40, $\Gamma \vdash \mathit{self} : B$, and by the Dictionary Rule 37, $\Gamma \vdash \varphi : B \Rightarrow A$. Thus, by the Substitution Lemma,

$$\Gamma \vdash [d \mapsto \varphi]\,[s \mapsto \mathit{self}]\,[\alpha \mapsto A]\,[\beta \mapsto B]\,e_{\varphi(\varphi'(l))} : \tau''$$

Since α occurs only covariantly in $\tau_{\varphi(\varphi'(l))}$ and $\Gamma \vdash A \preceq \tau_1'$, and β does not occur in $\tau_{\varphi(\varphi'(l))}$,

$$\Gamma \vdash [\alpha \mapsto A]\,[\beta \mapsto B]\,\tau_{\varphi(\varphi'(l))} \preceq [\alpha \mapsto \tau_1']\,\tau_{\varphi(\varphi'(l))}$$

Putting this together with the fact that $\tau = [\alpha \mapsto \tau_1']\,\tau_{\varphi(\varphi'(l))}$, we obtain

$$\Gamma \vdash [d \mapsto \varphi]\,[s \mapsto \mathit{self}]\,[\alpha \mapsto A]\,[\beta \mapsto B]\,e_{\varphi(\varphi'(l))} : \tau$$

as desired.

4. The rule used is the override rule, i.e.,

$$o \leftarrow_{\varphi'} n(\alpha, \beta, s, d, d') = e \rightsquigarrow o'$$

where

$$\begin{aligned}
o &= \mathsf{obj}(\alpha, \beta, s, d).\{\!|m \triangleright e_m : \tau_m \;^{m\in I}|\!\}_\varphi \\
o' &= \mathsf{obj}(\alpha, \beta, s, d).\{\!|m \triangleright e_m : \tau_m \;^{m\in I\setminus\varphi(\varphi'(n))},\ \varphi(\varphi'(n)) \triangleright [d'\mapsto\varphi]e : \tau'|\!\}_\varphi \\
\tau' &= \tau_\varphi(\varphi'(n))
\end{aligned}$$

By inspection the derivation of the first premise must end with the Override Rule 41, and involve a use of the Object Rule 40. Thus, there must be derivations

$$\begin{aligned}
&\Gamma \vdash \mathsf{obj}(\alpha, \beta, s, d).\{\!|m \triangleright e_m : \tau_m \;^{m\in I}|\!\}_\varphi : A \\
&\Gamma, \beta \preceq B, \alpha \preceq \top, d{:}\beta \Rightarrow \alpha, s{:}\beta \vdash e_m : \tau_m,\ (\forall m \in I) \\
&\Gamma \vdash A \preceq \tau \\
&\Gamma \vdash \varphi' : \tau \Rightarrow \mathsf{Obj}\,\alpha.\{\!|n : \tau'|\!\} \\
&\Gamma, \beta \preceq \top, \alpha \preceq \top, d'{:}\beta \Rightarrow \tau, d{:}\beta \Rightarrow \alpha, s{:}\beta \vdash e : \tau'
\end{aligned}$$

where

$$\begin{aligned}
A &= \mathsf{Obj}\,\alpha.\{\!|l : \tau_{\varphi(l)} \;^{l\in\mathsf{Dom}(\varphi)}|\!\} \\
B &= \mathsf{Obj}\,\alpha.\{\!|m : \tau_m \;^{m\in I}|\!\}
\end{aligned}$$

By bound weakening,

$$\Gamma, \beta \preceq B, \alpha \preceq \top, d'{:}\beta \Rightarrow \tau, d{:}\beta \Rightarrow \alpha, s{:}\beta \vdash e : \tau'$$

Note that $\Gamma, \beta \preceq B \vdash B \Rightarrow A \preceq \beta \Rightarrow \tau$, so by Subsumption,

$$\Gamma, \beta \preceq B \vdash \varphi : \beta \Rightarrow \tau$$

Thus, by Substitution,

$$\Gamma, \beta \preceq B, \alpha \preceq \top, d{:}\beta \Rightarrow \alpha, s{:}\beta \vdash [d \mapsto \varphi]\, e : \tau'$$

Therefore, the desired conclusion follows from the Object Rule 40.

5. The rule used is

$$\begin{aligned} o \leftarrow\!\!+ n(\alpha, \beta, s, d, d') = e : \tau' \\ \rightsquigarrow \mathsf{obj}(\alpha, \beta, s, d).\{\!|m \triangleright e_m : \tau\ ^{m \in I}, \\ n' \triangleright [d' \mapsto \varphi']\, e : \tau |\!\}_{\varphi'} \end{aligned}$$

where $n' = \mathsf{Fresh}(I)$ and

$$\begin{aligned} o &= \mathsf{obj}(\alpha, \beta, s, d).\{\!|m \triangleright e_m : \tau_m\ ^{m \in I}|\!\}_{\varphi} \\ \varphi' &= \varphi[n \mapsto n'] \end{aligned}$$

The derivation must end with a use of the Extension Rule 42, from which it follows that there must exist derivations

$$\begin{aligned} &\Gamma \vdash \mathsf{obj}(\alpha, \beta, s, d).\{\!|m \triangleright \tau_m : e_m\ ^{m \in I}|\!\}_{\varphi} : A \\ &\Gamma, \beta \preceq B, \alpha \preceq \top, d : \beta \Rightarrow \alpha, s{:}\beta \vdash e_m : \tau_m\ (\forall m \in I) \\ &\Gamma \vdash A \preceq \tau \\ &\Gamma, \beta \preceq \top, \alpha \preceq \top, d'{:}\beta \Rightarrow A', d{:}\beta \Rightarrow \alpha, s{:}\beta \vdash e : \tau' \end{aligned}$$

where

$$\begin{aligned} A &= \mathsf{Obj}\,\alpha.\{\!|l : \tau_{\varphi(l)}\ ^{l \in \mathsf{Dom}(\varphi)}|\!\} \\ \tau &= \mathsf{Obj}\,\alpha.\{\!|l : \tau_{\varphi(l)}\ ^{l \in J \subseteq \mathsf{Dom}(\varphi)}|\!\} \\ A' &= \mathsf{Obj}\,\alpha.\{\!|l : \tau_{\varphi(l)}\ ^{l \in \mathsf{Dom}(\varphi)}, n : \tau'|\!\} \\ B &= \mathsf{Obj}\,\alpha.\{\!|m : \tau_m\ ^{m \in I}|\!\} \end{aligned}$$

Let $B' = \mathsf{Obj}\,\alpha.\{\!|m : \tau_m\ ^{m \in I}, n' : \tau'|\!\}$. By Bound Weakening,

$$\Gamma, \beta \preceq B', \alpha \preceq \top, d : \beta \Rightarrow \alpha, s{:}\beta \vdash e_m : \tau_m$$

and

$$\Gamma, \beta \preceq B', \alpha \preceq \top, d'{:}\beta \Rightarrow A', d{:}\beta \Rightarrow \alpha, s{:}\beta \vdash e : \tau'$$

Note that $\Gamma, \beta \preceq B' \vdash B' \Rightarrow A' \preceq \beta \Rightarrow A'$, so by Subsumption,

$$\Gamma, \beta \preceq B' \vdash \varphi' : \beta \Rightarrow A'$$

Thus by substitution,

$$\Gamma, \beta \preceq B', \alpha \preceq T, d{:}\beta \Rightarrow \alpha, s{:}\beta \vdash [d \mapsto \varphi']e : \tau'$$

Therefore, the desired conclusion follows from the Object Rule 40.

This completes the case analysis and hence the proof. ■

LEMMA D.24 (Canonical Forms). *If* $\vdash v : (\tau \to \tau')$ *then* v *is a lambda expression. If* $\vdash v : \mathrm{Obj}\,\alpha.\{|m : \tau_m{}^{m\in I}|\}$ *then* v *is an object. If* $\vdash v : \tau \Rightarrow \tau'$ *then* v *is a dictionary.*

Proof. Direct from the typing rule for terms, given that subtyping in an empty context preserves the "shape" of types. ■

THEOREM D.25 (Progress). *If* $\vdash e : \tau$ *then either* e *is a value or else* $e \rightsquigarrow e'$.

Proof. The proof follows from Lemma D.24 and a comparison of expression forms with evaluation contexts. ■

ACKNOWLEDGMENTS

We thank Kathleen Fisher for several helpful conversations, Martin Odersky for pointing out the connections with Eiffel, and Viviana Bono, Perry Cheng, Gary Lindstrom, Joe Vanderwaart, and the anonymous referees for comments on drafts.

REFERENCES

1. Abadi, M., and Cardelli, L. (1996), "A Theory of Objects," Springer-Verlag, New York/Berlin.
2. Bono, V., Bugliesi, M., Dezani, M., and Liquori, L. (1997), Subtyping constraints for incomplete objects, *in* "CAAP," Lecture Notes in Computer Science, Springer-Verlag, New York/Berlin.
3. Bono, V., Bugliesi, M., and Liquori, L. (1996), A lambda calculus of incomplete objects, *in* "Proceedings, Mathematical Foundations of Computer Science," Lecture Notes in Computer Science, Vol. 1113, pp. 218–229. Springer-Verlag, New York/Berlin.
4. Bono, V., and Liquori, L. (1995), A subtyping for the Fisher–Honsell–Mitchell calculus of objects, *in* "Proceedings, Computer Science Logic 1994," Lecture Notes in Computer Science, Vol. 933, pp. 16–30, Springer-Verlag, New York/Berlin.
5. Bracha, G., and Lindstrom, G. (1992), Modularity meets inheritance, *in* "Proceedings of the IEEE Computer Society International Conference on Computer Languages," pp. 282–290, IEEE Computer Society, Washington, D.C.
6. Bruce, K. B. (1994), A paradigmatic object-oriented language: Design, static typing, and semantics, *J. Functional Programming* **4**(2), 127–206.
7. Bruce, K. B., Cardelli, L., Castagna, G., The Hopkins Object Group, Leavens, G., and Pierce, B. C. (1995), On binary methods, *Theory and Practice of Object Systems* **1**(3), 217–238.
8. Cardelli, L., and Wegner, P. (1985), On understanding types, data abstraction and parametric polymorphism, *Computing Surveys* **17**(4), 471–522.
9. Drossopoulou, S., Wragg, D., and Eisenbach, S. (1998), What *is* Java binary compatibility? —Version 2, available from http://www-dse.doc.ic.ac.uk in the file projects/slurp/papers.html#bchuge.
10. Felleisen, M. (1988), The theory and practice of first-class prompts, *in* "Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages," pp. 180–190, ACM.
11. Fisher, K. (1996), "Type Systems for Object-Oriented Programming Languages," Ph.D. thesis, Department of Computer Science, Stanford University, 1996.
12. Fisher, K., Honsell, F., and Mitchell, J. C. (1994), A lambda calculus of objects and method specialization, *Nordic J. Comput.* (formerly BIT), **1**, 3–37 [1993, a preliminary version appeared in "Proc. IEEE Symposium on Logic in Computer Science," pp. 26–38, IEEE].
13. Fisher, K., and Mitchell, J. C. (1995), A delegation-based object calculus with subtyping, *in* "Fundamentals of Computation Theory (FCT'95)," Lecture Notes in Computer Science, Vol. 965, pp. 42–61, Springer-Verlag, New York/Berlin.
14. Fisher, K., and Mitchell, J. C. (1996), The development of type systems for object-oriented languages, *Theory and Practice of Object Systems*, **1**, 189–220 [1994, a preliminary version appeared in "Proc. Theoretical Aspects of Computer Software," Lecture Notes in Computer Science, Vol. 789, pp. 844–885].
15. Fisher, K., and Mitchell, J. C. (1997), On the relationship between classes, objects, and data abstraction, *in* "Proceedings of the International Summer School on Mathematics of Program Construction, Marktoberdorf, Germany," Lecture Notes in Computer Science. Springer-Verlag, New York/Berlin to appear [revised version to appear in "Theory and Practice of Object Systems"].
16. Fisher, K., and Reppy, S. H., MOBY objects and classes, unpublished manuscript, 1998.

17. Jategaonkar, L., and Mitchell, J. C. (1993), Type inference with extended pattern matching and subtypes, *Fundamenta Informaticae* **19**, 127–166 [1988, a preliminary version appeared in "Proceedings of the ACM Symposium on Lisp and Functional Programming"].
18. Lakos, J. (1996), "Large-Scale C++ Software Design," Addison–Wesley, Reading, MA, 1996.
19. Liquori, L. (1996), An extended theory of primitive objects: First and second order systems, Tech. Rep. CS-23-96, Dipartimento di Informatica, Università di Torino.
20. Liquori, L. (1997), An extended theory of primitive objects: First order system, *in* suoka, editors, "Proceedings of ECOOP-97, International European Conference on Object Oriented Programming" (M. Aksit and S. Matsuoka, Eds.), Lecture Notes in Computer Science, Vol. 1241, Springer-Verlag, New York/Berlin.
21. Meyer, B. (1992), "Eiffel: The Language," Prentice–Hall, Englewood Cliffs, NJ.
22. Mitchell, J. C. (1990), Toward a typed foundation for method specialization and inheritance, *in* "Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages," pp. 109–124. ACM.
23. Mitchell, J. C., and Plotkin, G. D. (1988), Abstract types have existential type, *ACM Trans. Programming Languages and Systems* **10**(3), 470–502.
24. Plotkin, G. D. (1981), A structural approach to operational semantics, Tech. Rep. DAIMI FN-19, Computer Science Dept., Aarhus Univ., Denmark.
25. Rémy, D. (1994), Type inference for records in a natural extension of ML, *in* "Theoretical Aspects of Object-Oriented Programming" (C. A. Gunter and J. C. Mitchell, Eds.), pp. 67–95, MIT Press, Cambridge, MA [1989, an earlier version appeared *in* "Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages," ACM].
26. Rémy, D., and Vouillon, J. (1997), Objective ML: A simple object-oriented extension of ML, *in* "Proceedings of the 24th ACM Symposium on Principles of Programming Languages," pp. 40–53, ACM Press.
27. Riecke, J. G., and Stone, C. A. (1998), Privacy via subsumption, *in* "Informal Workshop Record of the Fifth Workshop on Foundations of Object-Oriented Languages."
28. Wand, M. (1987), Complete type inference for simple objects, *in* "Proceedings, Symposium on Logic in Computer Science," pp. 37–44, IEEE.