Privacy via Subsumption

Jon G. Riecke
Bell Laboratories
Lucent Technologies
700 Mountain Avenue
Murray Hill, NJ 07974 USA
riecke@bell-labs.com

Christopher A. Stone
School of Computer Science
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213 USA
cstone@cs.cmu.edu

Abstract

We describe an object calculus that allows both extension of objects and full width subtyping (hiding arbitrary components). In contrast to other proposals, the types of our calculus do not mention "missing" methods. To avoid type unsoundness, the calculus mediates all interaction with objects via "dictionaries" that resemble the method dispatch tables in conventional implementations. Private fields and methods can be modeled and enforced by scoping restrictions: forgetting a field or method through subsumption makes it private. We prove that the type system is sound, give a variant which allows covariant self types, and give some examples of the expressiveness of the calculus.

1 Introduction

One of the most important principles of software engineering is information hiding: the ability to build and enforce data or procedural abstractions in order to make programs readable and maintainable. Most object-oriented programming languages provide direct support for information hiding. Class-based languages like C++ and Java [14], for instance, have mechanisms for hiding methods and fields via **private** annotations; private methods and fields added to an object may be accessed only by other methods defined within the same class.

In this paper we give an elementary account of private fields and methods in the presence of structural subtyping. We extend recent object calculi [1, 20] with operations for hiding and renaming methods, operations that can also be found in the class system of Eiffel [19]. The primitives of the 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 prove that the type systems prevent run-time type errors.

We take the somewhat novel approach of distinguishing between internal and external names of methods. In some sense, common implementations of object-oriented languages already use this idea: classes with methods are compiled into a table, with a pointer to each method, and the internal names are their location within the table. Our calculus distinguishes internal from external names so that methods can be made private, even after an object or class has been created. Once the external name of the method has been lost (either through an explicit operation or implicitly by subsumption), the method cannot be overridden; other methods referring to this method by its internal name 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 problem with object extension. In previous formalizations of extensible objects, the combination of object extension and unrestricted width subtyping was unsound. For instance, suppose we ignore this distinction and define an object p containing two methods, x which always returns 3, and $getx$ which returns the value of the x method:

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

The variable s stands for "self", and is dynamically bound to the object upon method invocation. The type of p is $\{\!|x : \mathsf{Int}, getx : \mathsf{Int}|\!\}$. If p can also have the less precise type $\{\!|getx : \mathsf{Int}|\!\}$, there is be nothing to prevent us from adding a new method x returning the value True of type Bool. In the dynamic semantics of [10, 18] where 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; This is a type error as $getx$ is statically typed as returning an integer. To avoid such errors, the type systems of [10, 18] weaken the subtyping relation for extensible objects: either methods may not be hidden at all, or components can be made "inaccessible" (cannot be invoked) though still visible and overridable. 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. 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 re-typechecked and possibly recompiled (a problem familiar from C++ [16]). In the worst case, derived classes become ill-formed and large pieces of code must be rewritten.

We approach this problem by allowing unrestricted hiding and compensating in the dynamic semantics. 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.

2 A Language of 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

Table 1: Syntax of First-Order System

$\tau ::=$	b	base type
$\mid$	$(\tau{\to}\tau')$	function type
$\mid$	$\{\!\mid l : \tau_l \,^{l\in I} \mid\!\}$	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
$\mid$	$\mathsf{obj}\, s.\{\!\mid i \triangleright e_i : \tau_i \,^{i\in 1..n} \mid\!\}_\varphi$	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

calculus to a simple delegation-based system; objects, being extensible, also serve to encode classes. Variants of the calculus have been carefully studied before (e.g., [9, 10, 11, 12, 18]). 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.

2.1 Syntax

The language, whose syntax appears in Table 1, derives largely from the object calculi of Abadi and Cardelli [1], Fisher, Honsell, and Mitchell [10], and Liquori [18]. The types of the language include base types, function types, and object types. Object types only mention the visible names of methods and their return types. We identify object expressions or types differing only in the order of their components.

In most object calculi, objects draw their method names from a distinguished infinite set L of labels. Instead, our calculus distinguishes *internal* names from *external* names, and uses a dictionary to map between the two. For convenience, we choose the natural numbers $\mathbf{N}$ to serve as the internal names, L to serve as external names, and use l or m to denote elements of the set $L \cup \mathbf{N}$. The use of the natural numbers is meant to evoke "slots" in a method table or array, and makes the notation slightly simpler. There is, of course, nothing essential in using $\mathbf{N}$, or even in distinguishing the sets of external and internal labels.

A **dictionary** φ is a finite partial function from the extended set of labels $L \cup \mathbf{N}$ to $L \cup \mathbf{N}$. Dictionaries appear as subscripts on objects. For instance, the object

$$\mathsf{obj}\, s.\{\!\mid 1 \triangleright 3 : \mathsf{Int} \mid\!\}_{[x \mapsto 1]}$$

has the dictionary $[x \mapsto 1]$; when x is invoked, the actual code invoked is the method internally labeled by $[x{\mapsto}1](x) = 1$. We use $\varphi[l \mapsto n]$ to denote the partial function that behaves exactly as φ except for mapping l to n, and $id(S)$ to denote the identity function with finite domain $S \subseteq L \cup \mathbf{N}$.

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,

$$\mathsf{obj}\, s.\{\!\mid 1 \triangleright 3 : \mathsf{Int}, 2 \triangleright (s@\varphi).x : \mathsf{Int} \mid\!\}_\varphi @[getx \mapsto getx]$$

where $\varphi = [x \mapsto 1, getx \mapsto 2]$, returns an object whose only visible method is $getx$, because the dictionary will be $[getx \mapsto 2]$. Similarly, one can increase the number of fields visible by mapping several external labels to the same internal label. (In this case, if one of these methods is overridden multiple methods may appear to change.) The other two operations add or change the methods of objects. 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. An existing method can be replaced within an object by the operation $e_0{\leftarrow}l(s){=}e$.

2.2 Static and dynamic semantics

To give dynamic semantics to the language, we use Felleisen's "evaluation context" formulation [8] of Plotkin's SOS [21]. The syntax of evaluation contexts (a subset of those expressions containing a single hole, denoted $[\cdot]$) is given by the grammar

$$\begin{array}{rl} E ::= & [\cdot] \mid (E\ e) \mid (v\ E) \\ \mid & E.l \\ \mid & E@\varphi \\ \mid & E{\leftarrow\!+}l(s){=}e' : \tau' \\ \mid & E{\leftarrow}l(s){=}e' \end{array}$$

We write $E[e]$ to denote the evaluation context E with the hole replaced by e. The local reduction relation $\leadsto$ is shown in Table 2. These rules use a syntactic substitution operation, written $[s{\mapsto}e]\,e'$, which denotes the capture-free substitution of e for s in e'. The important point to note for this system is that the dictionary is stripped (and replaced by an identity dictionary) at method invocation. This allows method code to assume a default dictionary, however the object may be altered. Code for methods in method override and object extension is then coerced to expect such stripped objects. The relation in Table 2 is extended to a one-step evaluation relation on programs by

$$e \leadsto e' \quad \Longleftrightarrow \quad \exists e_1, e_2.\ e = E[e_1] \ \wedge\ e_1 \leadsto e_2 \ \wedge\ E[e_2] = e'.$$

We can prove

Proposition 1 (Determinacy)
The relation $\leadsto$ *is a partial function.*

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

The static semantics of the language is shown in Appendix A. The novel aspects are the subsumption rule for width subtyping—more precisely the fact that we allow naïve width subtyping on objects—and the treatment of dictionaries. The following theorem shows that the static semantics prevents run-time type errors:

Theorem 2 (Soundness)

1. *If* $\vdash e : \tau$ *and* $e \leadsto e'$, *then* $\vdash e' : \tau$.
2. *If* $\vdash e : \tau$, *then* e *is a value or* $e \leadsto e'$ *for some expression* e'.

2.3 Examples

The first example shows the behavior of private methods. Let $\varphi = [F{\mapsto}1, M{\mapsto}2]$ and define the explicit subtyping coercion $o{:}{>}\tau$ as

Table 2: Local Reduction Steps of First-Order System

$(\lambda x{:}\tau.e)\ v$	$\leadsto$	$[x\mapsto v]\,e$	
$(\mathsf{obj}\,s.\{\!\|i\triangleright e_i : \tau_i\ ^{i\in 1..n}\|\!\}_{\varphi}).l$	$\leadsto$	$[s\mapsto \mathsf{obj}\,s.\{\!\|i\triangleright e_i : \tau_i\ ^{i\in 1..n}\|\!\}_{id\{1,\ldots,n\}}]\,e_{\varphi(l)}$	
$(\mathsf{obj}\,s.\{\!\|i\triangleright e_i : \tau_i'\ ^{i\in 1..n}\|\!\}_{\varphi})@\varphi'$	$\leadsto$	$\mathsf{obj}\,s.\{\!\|i\triangleright e_i : \tau_i'\ ^{i\in 1..n}\|\!\}_{\varphi\circ\varphi'}$	
$(\mathsf{obj}\,s.\{\!\|i\triangleright e_i : \tau_i\ ^{i\in 1..n}\|\!\}_{\varphi})\leftarrow\!\!+l(s){=}e : \tau$	$\leadsto$	$\mathsf{obj}\,s.\{\!\|i\triangleright e_i : \tau_i\ ^{i\in 1..n}, (n+1)\triangleright[s\mapsto s@\varphi']\,e : \tau\|\!\}_{\varphi'}$	$\varphi' = \varphi[l\mapsto(n{+}1)]$
$(\mathsf{obj}\,s.\{\!\|i\triangleright e_i : \tau_i\ ^{i\in 1..n}\|\!\}_{\varphi})\leftarrow l(s){=}e$	$\leadsto$	$\mathsf{obj}\,s.\{\!\|i\triangleright e_i : \tau_i\ ^{i\in(1..n)\backslash\varphi(l)}, \varphi(l)\triangleright[s\mapsto s@\varphi]\,e : \tau_{\varphi(l)}\|\!\}_{\varphi}$	

shorthand for the term $((\lambda x{:}\tau.x)\ o)$. Consider the terms

$$
\begin{aligned}
&o := ((\mathsf{obj}\,s.\{\!\|\|\!\}_{[]})\leftarrow\!\!+F(s){=}5 : \mathsf{Int})\leftarrow\!\!+M(s){=}(s.F+1) : \mathsf{Int}\\
&\qquad o : \{\!\|F : \mathsf{Int}, M : \mathsf{Int}\|\!\}\\
&\qquad o \leadsto^{*} \mathsf{obj}\,s.\{\!\|1\triangleright 5 : \mathsf{Int}, 2\triangleright(s@\varphi).F+1 : \mathsf{Int}\|\!\}_{\varphi}\\
&o_1 := o\leftarrow F(s){=}7\\
&\qquad o_1 : \{\!\|F : \mathsf{Int}, M : \mathsf{Int}\|\!\}\\
&\qquad o_1 \leadsto^{*} \mathsf{obj}\,s.\{\!\|1\triangleright 7 : \mathsf{Int}, 2\triangleright(s@\varphi).F+1 : \mathsf{Int}\|\!\}_{\varphi}\\
&o_2 := o{:>}\{\!\|M : \mathsf{Int}\|\!\}\\
&\qquad o_2 : \{\!\|M : \mathsf{Int}\|\!\}\\
&\qquad o_2 \leadsto^{*} \mathsf{obj}\,s.\{\!\|1\triangleright 7 : \mathsf{Int}, 2\triangleright(s@\varphi).F+1 : \mathsf{Int}\|\!\}_{\varphi}\\
&o_3 := o_2\leftarrow\!\!+F(s){=}\mathsf{True} : \mathsf{Bool}\\
&\qquad o_3 : \{\!\|F : \mathsf{Bool}, M : \mathsf{Int}\|\!\}\\
&\qquad o_3 \leadsto^{*} \mathsf{obj\,s}.\{\!\|1\triangleright 7 : \mathsf{Int}, 2\triangleright(s@\varphi).F+1 : \mathsf{Int},\\
&\qquad\qquad\qquad 3\triangleright\mathsf{True} : \mathsf{Bool}\|\!\}_{[F\mapsto 3, M\mapsto 2]}
\end{aligned}
$$

Here o 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 $o.F$ evaluates to 5 and $o.M$ 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 6. 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 6, while $o_3.F$ returns True.

As this example shows, under our semantics 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

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

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 so that o_4 evaluates to the same object as o_3. 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.

As a second example, consider the term

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

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. An extension of the system with row variables [24] or bounded polymorphism [7] might allow preservation of methods in such cases.

3 Second-Order System

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:

$$PT' := \{\!\|getx : \mathsf{Int}, move : \mathsf{Int}{\to}PT'\|\!\}$$

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

$$CPT' := \{\!\|getx : \mathsf{Int}, move : \mathsf{Int}{\to}PT', getc : \mathsf{color}\|\!\}$$

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

This motivates a move to a "second-order calculus" in the parlance of [1]. Method types can now refer to "the type of self" which changes as the object is extended. Thus we define

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

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

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

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

Table 3: Syntax of the Second-Order System.

$\tau ::=$	b	base type
$\mid$	α	type variable
$\mid$	$\tau\to\tau'$	function type
$\mid$	$\tau\Rightarrow\tau'$	dictionary type
$\mid$	$\mathsf{Obj}\,\alpha.\{\!\vert l:\tau_l{}^{l\in I}\vert\!\}$	object type
$\mid$	$\mathsf{Obj}\,\alpha.\{\!\vert i:\tau_i{}^{i\in 1..n}\vert\!\}_\varphi$	layout type
$\Gamma ::=$	$\bullet$	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
$\mid$	φ	dictionary
$\mid$	$\mathsf{obj}\,(s{:}\alpha).\{\!\vert i\triangleright e_i:\tau_i'{}^{i\in 1..n}\vert\!\}_\varphi$	object
$e ::=$	v	value
$\mid$	$e_1\ e_2$	application
$\mid$	$e.l$	method invocation
$\mid$	$e\,\&\,v$	dictionary replacement
$\mid$	$e\leftarrow\!\!+l(s{:}\alpha, d)=e':\tau$	object extension
$\mid$	$e\leftarrow l(s{:}\alpha, d)=e'$	method override
$\mid$	$e._v l$	invocation with dictionary
$\mid$	$e\leftarrow_v l(s{:}\alpha, d)=e'$	override with dictionary

3.1 Syntax and static semantics

In parallel with the extension of syntax for object types, we also extend object values to bind a type variable α to represent the type of self within the object:

$$\mathsf{obj}\,(s{:}\alpha).\{\!|i\triangleright e_i:\tau_i{}^{i\in 1..n}|\!\}_\varphi$$

(The entire syntax of the second-order language appears in Table 3.) Because of the possibility of object extension and width subtyping, there is no way to know when a method is typechecked exactly what the type of self will be when the method is invoked. Therefore, the standard technique for typechecking methods involving such self types is to require they be parametric in the type of *self*, so that they are guaranteed to work for *any* future version of the object; normally this is implemented by using bounded polymorphism.

Because we have dictionaries, we have a further complication: methods involving self types must work parametrically not only for any extension of the object, but for any change to the object's dictionary as well. For example, the code for *pt.move* and *cpt.move* above is the same by construction; however, *pt* and *cpt* will have different dictionaries, and the values returned by the *move* functions should have similarly different dictionaries. This causes difficulties for our first-order system, in which invoking a method discards the object's dictionary. (Another way of looking at the first-order semantics is that all methods added by extension or override start by replacing the dictionary on *self* with their own dictionary.)

The second-order calculus avoids such coercions by providing dictionaries as first-class values, and allowing operations to be *parameterized* by a dictionary. Both method lookup and method override can specify the particular dictionary to be used in finding the internal name for the method. Thus the code for the *move* method above will be able to access or override the *getx* method of *self* by using the dictionary appropriate for points *without* modifying the dictionary of *self*; *pt.move* will return a point and *cpt.move* will return a colored point.

We do not expect programmers to build dictionaries directly; the current dictionary of an object is supplied for use in bodies at method override or method extension time. As seen in Table 3, the syntax for method override and method extension binds not only a variable s representing *self* and a type variable α representing self type, but also a variable d which represents the dictionary of the object at the time the method is added. Method override and method invocation also may optionally specify a dictionary value. For example, the operation $e\leftarrow_v l(s{:}\alpha, d)=e'$ uses the dictionary v to find the method that needs to be replaced (namely, the one with internal label $v(l)$). In general, v is likely to be either a variable d or a constant dictionary.

The last operation we provide is dictionary replacement, written $e\,\&\,\varphi$, since there are occasions in which we still want to replace the dictionary of an object entirely. For simplicity, we do not provide the dictionary renaming operation of the first-order system, although it would not be difficult to add.

Not all dictionaries make sense in conjunction with all objects. If all one knows is the external names of an object, it is impossible to tell whether a particular dictionary is appropriate for that object. Therefore we extend the type system with **dictionary types** (written with $\Rightarrow$), and **layout types**. Layout types are object types exposing the inner layout of an object and information about its dictionary. The most precise type of an object is a layout type giving the types and internal names for all its components, and exposing the entries in its dictionary. Every layout type is a subtype of the corresponding object type derived from the component types and the dictionary specification. For instance, the layout type

$$\mathsf{Obj}\,\alpha.\{\!|1:\mathsf{Int}, 2:\mathsf{Int}\to\alpha|\!\}_{[x\mapsto 1, move\mapsto 2]}$$

is a subtype of the object type

$$\mathsf{Obj}\,\alpha.\{\!|x:\mathsf{Int}, move:\mathsf{Int}\to\alpha|\!\}.$$

An object type is thus more abstract than any of its possible layout types.

Layout types are used in conjunction with dictionary types; a dictionary of type $\tau_1\Rightarrow\tau_2$ can be used with objects of type τ_1 (usually a layout type or a type variable representing a layout type), and when used to replace that object's dictionary yields an object of type τ_2. In the second-order system, we can restrict dictionaries to finite, partial functions from L to $\mathbf{N}$.

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

$$\top_{obj} := \mathsf{Obj}\,\alpha.\{\!||\!\}$$
$$(\mathsf{let}\,x:\tau = e\,\mathsf{in}\,e') := ((\lambda x{:}\tau.e')\ e)$$

Here $\top_{obj}$ is the object type conveying the least information.

The typing rules resemble those of the first-order system, but with two significant additions. First, since the type of *self* is now a type variable, we need assumptions about the bounds of type variables. A new rule for this appears in the rules for well-formedness of contexts. Second, the system includes rules for the new operations and values. The rules for checking objects, method override, and method extension are the most complicated. These rules check the bodies of methods using assumptions about the type of *self* and (possibly) a dictionary d. In contrast to other systems in which the type α of *self* is a type variable bounded by some object type τ, we assume α is an abstract object type, and supply a dictionary d of type $\alpha\Rightarrow\tau$. This corresponds to the idea that in order to make real use of *self* as an object, we have to use a known dictionary rather than the one currently attached to *self*. For methods within object values, we have information about the underlying layout and hence can make self type a type variable bounded by a layout type.

A technical constraint is that the type α of *self* must appear *covariantly* inside layout and 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;
- τ is $\mathsf{Obj}\,\alpha.\{\!| l : \tau_l\ ^{l \in I} |\!\}$ or $\mathsf{Obj}\,\alpha.\{\!| i : \tau_i\ ^{i \in 1..n} |\!\}_\varphi$, and α occurs covariantly in each τ_i.

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;
- τ is $\mathsf{Obj}\,\alpha.\{\!| l : \tau_l\ ^{l \in I} |\!\}$ or $\mathsf{Obj}\,\alpha.\{\!| i : \tau_i\ ^{i \in 1..n} |\!\}_\varphi$, and α occurs contravariantly in each τ_i.

We would need more restrictive width subtyping to avoid unsoundness if the α were allowed to appear contravariantly (see [1] for examples). As such, this system does not handle binary methods (see [6] for a thorough discussion).

3.2 Dynamic semantics

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

$$\begin{array}{rll} E ::= & [\cdot] \mid (E\ e) \mid (v\ E) \\ \mid & E \,\&\, v \\ \mid & E.l \\ \mid & E._v l \\ \mid & E \leftarrow\!\!+ l(s{:}\alpha, d) = e' : \tau' \\ \mid & E \leftarrow l(s{:}\alpha, d) = e' \\ \mid & E \leftarrow_v l(s{:}\alpha, d) = e' \end{array}$$

The rules for reducing redexes appear in Table 4; these rules define a relation $\leadsto$. As with the dynamic semantics of the first-order system, we write $e \leadsto e'$ when there is an evaluation context E such that $e = E[e_1]$, $e_1 \leadsto e_2$, and $e' = E[e_2]$. The dynamic semantics is deterministic:

Proposition 3 (Determinacy)
The relation $\leadsto$ is a partial function.

We again use the relation $\leadsto^*$ to denote the reflexive, transitive closure of $\leadsto$.

The static semantics is sound for the dynamic semantics:

Theorem 4 (Soundness)
1. *If $\vdash e : \tau$ and $e \leadsto e'$, then $\vdash e' : \tau$.*
2. *If $\vdash e : \tau$, then e is a value or $e \leadsto e'$ for some expression e'.*

3.3 Examples

Standard examples can now be written in the calculus. For instance, we can write code matching the above point and color point types:

$$\begin{array}{rl} o & := (\mathsf{obj}\,(s{:}\alpha).\{\!|\,|\!\}) \leftarrow\!\!+ getx(s{:}\alpha, d) = 0 : \mathsf{Int} \\ pt & ::= o \leftarrow\!\!+ move(s : \alpha, d) = \\ & \qquad \lambda y{:}\mathsf{Int}.\mathsf{let}\ z = s._d getx \\ & \qquad\qquad \mathsf{in}\ s \leftarrow_d getx(s'{:}\alpha', d') = z + y \\ cpt & ::= pt \leftarrow\!\!+ getc(s{:}\alpha, d) = \mathsf{Red} : \mathsf{Color} \end{array}$$

There are many possible types for these objects besides PT and CPT. The most specific types are the layout types

$$\begin{array}{rl} o : & \mathsf{Obj}\,\alpha.\{\!| 1 : \mathsf{Int} |\!\}_{[getx \mapsto 1]} \\ pt : & \mathsf{Obj}\,\alpha.\{\!| 1 : \mathsf{Int}, 2 : \mathsf{Int} \to \alpha |\!\}_{[getx \mapsto 1, move \mapsto 2]} \\ cpt : & \mathsf{Obj}\,\alpha.\{\!| 1 : \mathsf{Int}, 2 : \mathsf{Int} \to \alpha, 3 : \mathsf{Color} |\!\}_{[getx \mapsto 1, move \mapsto 2, getc \mapsto 3]} \end{array}$$

These types are probably too detailed to report to the programmer, and might be suppressed by a real system. Instead, the following types are the most specific object types:

$$\begin{array}{rl} o : & \mathsf{Obj}\,\alpha.\{\!| getx : \mathsf{Int} |\!\} \\ pt : & \mathsf{Obj}\,\alpha.\{\!| getx : \mathsf{Int}, move : \mathsf{Int} \to \alpha |\!\} = PT \\ cpt : & \mathsf{Obj}\,\alpha.\{\!| getx : \mathsf{Int}, move : \mathsf{Int} \to \alpha, c : \mathsf{Color} |\!\} = CPT \end{array}$$

These types are supertypes of the above layout types.

The dynamic semantics of the language reduces these expressions to the following values:

$$\begin{array}{rl} o \leadsto^* & \mathsf{obj}\,(s{:}\alpha).\{\!| 1 \triangleright 0 : \mathsf{Int} |\!\}_{[getx \mapsto 1]} \\ pt \leadsto^* & \mathsf{obj}\,(s{:}\alpha).\{\!| 1 \triangleright 0 : \mathsf{Int}, \\ & \quad 2 \triangleright (\lambda y{:}\mathsf{Int}.\mathsf{let}\ z = s._\varphi getx \\ & \qquad\qquad \mathsf{in}\ s \leftarrow_\varphi getx(s'{:}\alpha', d') = z + y) : \\ & \qquad \mathsf{Int} \to \alpha \\ & \quad |\!\}_\varphi \\ cpt \leadsto^* & \mathsf{obj}\,(s{:}\alpha).\{\!| 1 \triangleright 0 : \mathsf{Int}, \\ & \quad 2 \triangleright (\lambda y{:}\mathsf{Int}.\mathsf{let}\ z = s._\varphi getx \\ & \qquad\qquad \mathsf{in}\ s \leftarrow_\varphi getx(s'{:}\alpha', d') = z + y) : \\ & \qquad \mathsf{Int} \to \alpha, \\ & \quad 3 \triangleright \mathsf{Red} : \mathsf{Color} \\ & \quad |\!\}_{[getx \mapsto 1, move \mapsto 2, getc \mapsto 3]} \end{array}$$

where $\varphi = [getx \mapsto 1, move \mapsto 2]$. Notice that the dictionary used in the body of the *move* method is φ, the one in place when the *move* method was added to the object. Thus, the *move* method looks in slot 1 for *getx*, and updates slot 1, even if a new method *getx* is later added to the object. The return type of *move* is still the type of *self*, so invoking *move* from both *pt* and *cpt* returns an object of the right type.

We can go a step further and define a form of classes for points and colored points, here instantiated as object-generating functions. For example,

$$\begin{array}{rl} pt\_class := & \lambda(x_0 : \mathsf{Int}). \\ & \quad (\mathsf{obj}\,(s{:}\alpha).\{\!|\,|\!\}_{[]} \\ & \quad \leftarrow\!\!+ x(s{:}\alpha, d) = x_0 : \mathsf{Int} \\ & \quad \leftarrow\!\!+ getx(s{:}\alpha, d) = s._d x : \mathsf{Int} \\ & \quad \leftarrow\!\!+ move(s{:}\alpha, d) = \\ & \qquad \lambda y{:}\mathsf{Int}.\mathsf{let}\ z = s._d getx \\ & \qquad\qquad \mathsf{in}\ s \leftarrow_d x(s'{:}\alpha', d') = z + y \\ & \quad) :> PT \end{array}$$

$$\begin{array}{rl} cpt\_class := & \lambda(x_0{:}\mathsf{Int}).\lambda(c_0{:}\mathsf{Color}). \\ & \quad (pt\_class(x_0) \\ & \quad \leftarrow\!\!+ c(s : \alpha, d) = c_0 : \mathsf{Color} \\ & \quad \leftarrow\!\!+ getc(s{:}\alpha, d) = s._d c : \mathsf{Int}) :> CPT \end{array}$$

Here we have added the private field x, which is used by *getx* and *move*, but hidden from view by subsumption; *cpt_class* has type $\mathsf{Int} \to PT$. In particular, x is not in the scope of methods in the class *cpt_class* inheriting from *pt_class*. Similarly, we have added a private field c to the class of colored points, accessible only by the *getc* method, but through coercion the type of *cpt_class* is $\mathsf{Int} \to \mathsf{Color} \to CPT$. To typecheck and compile *cpt_class* we need only know the above type of *pt_class* which does not expose x.

Table 4: Local Reduction Steps for Second-Order System.

$(\lambda x{:}\tau.e)\ v$	$\leadsto$	$[x\mapsto v]\,e$	
$(\mathsf{obj}\,(s{:}\alpha).\{\vert i\triangleright e_i : \tau_i\ ^{i\in 1..n}\vert\}_{\varphi}).l$	$\leadsto$	$[s\mapsto \mathsf{obj}\,(s{:}\alpha).\{\vert i\triangleright e_i : \tau_i\ ^{i\in 1..n}\vert\}_{\varphi}]\,[\alpha\mapsto A]\,e_{\varphi(l)}$	$A = \mathsf{Obj}\,\alpha.\{\vert i : \tau_i\ ^{i\in 1..n}\vert\}_{\varphi}$
$(\mathsf{obj}\,(s{:}\alpha).\{\vert i\triangleright e_i : \tau_i\ ^{i\in 1..n}\vert\}_{\varphi}).{}_{\varphi'}l$	$\leadsto$	$[s\mapsto \mathsf{obj}\,(s{:}\alpha).\{\vert i\triangleright e_i : \tau_i\ ^{i\in 1..n}\vert\}_{\varphi}]\,[\alpha\mapsto A]\,e_{\varphi'(l)}$	$A = \mathsf{Obj}\,\alpha.\{\vert i : \tau_i\ ^{i\in 1..n}\vert\}_{\varphi}$
$(\mathsf{obj}\,(s{:}\alpha).\{\vert i\triangleright e_i : \tau_i\ ^{i\in 1..n}\vert\}_{\varphi})\,\&\,\varphi'$	$\leadsto$	$\mathsf{obj}\,(s{:}\alpha).\{\vert i\triangleright e_i : \tau_i\ ^{i\in 1..n}\vert\}_{\varphi'}$	
$(\mathsf{obj}\,(s{:}\alpha).\{\vert i\triangleright e_i : \tau_i\ ^{i\in 1..n}\vert\}_{\varphi})\leftarrow l(s{:}\alpha,d)=e$	$\leadsto$	$\mathsf{obj}\,(s{:}\alpha).\{\vert i\triangleright e_i : \tau_i\ ^{i\in(1..n)\setminus\varphi(l)}, \varphi(l)\triangleright[d\mapsto\varphi]\,e : \tau_{\varphi(l)}\vert\}_{\varphi}$	
$(\mathsf{obj}\,(s{:}\alpha).\{\vert i\triangleright e_i : \tau_i\ ^{i\in 1..n}\vert\}_{\varphi})\leftarrow_{\varphi'} l(s{:}\alpha,d)=e$	$\leadsto$	$\mathsf{obj}\,(s{:}\alpha).\{\vert i\triangleright e_i : \tau_i\ ^{i\in(1..n)\setminus\varphi'(l)}, \varphi'(l)\triangleright[d\mapsto\varphi]\,e : \tau_{\varphi'(l)}\vert\}_{\varphi}$	
$(\mathsf{obj}\,(s{:}\alpha).\{\vert i\triangleright e_i : \tau_i\ ^{i\in 1..n}\vert\}_{\varphi})\leftarrow\!+\,l(s{:}\alpha,d)=e:\tau$	$\leadsto$	$\mathsf{obj}\,(s{:}\alpha).\{\vert i\triangleright e_i : \tau_i\ ^{i\in 1..n}, (n{+}1)\triangleright[d\mapsto\varphi']\,e : \tau\vert\}_{\varphi'}$	$\varphi' = \varphi[l\mapsto(n{+}1)]$

This series of examples did not use dictionary replacement; this operation is useful when we want to delegate part of the action of a method to a predeclared function. For instance, define the function *getf* by:

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

Then define the objects

$$\begin{aligned} o_1 :=\ & \mathsf{obj}\,(s{:}\alpha).\{\vert\vert\} \\ & \leftarrow\!+F(s{:}\alpha,d){=}4 : \mathsf{Int} \\ & \leftarrow\!+M(s{:}\alpha,d){=}getf(s\,\&\,d) : \mathsf{Int} \end{aligned}$$

$$o_1 : \mathsf{Obj}\,\alpha.\{\vert F : \mathsf{Int}, M : \mathsf{Int}\vert\}$$

$$\begin{aligned} o_2 :=\ & o_1 \\ & \leftarrow\!+F(s{:}\alpha,d){=}5 : \mathsf{Int} \\ & \leftarrow\!+N(s{:}\alpha,d){=}getf(s\,\&\,d) : \mathsf{Int} \end{aligned}$$

$$o_2 : \mathsf{Obj}\,\alpha.\{\vert F : \mathsf{Int}, M : \mathsf{Int}, N : \mathsf{Int}\vert\}$$

Then we have

$$\begin{array}{ll} o_1.F \leadsto^* 4 & o_2.F \leadsto^* 5 \\ o_1.M \leadsto^* 4 & o_2.M \leadsto^* 4 \\ & o_2.N \leadsto^* 5 \end{array}$$

Although $o_2.M$ and $o_2.N$ appear to have the same code, they evaluate to different values because the dictionaries they use have different views of which field corresponds to F. Recall also that although o_2 has an object type with three components, because we used extension rather than override for the second F field, the underlying representation has four methods.

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 methods 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 would be unsound to drop object components when they are hidden by subsumption, it would be 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 makes a good 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.

Calls to methods from *outside* the method suite may still need to go through the dictionary, however. In Java, for instance, suppose we define two classes `A` and `B` and an interface `C` 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.

4.2 Related work

Our calculus embodies solutions to two problems: providing a characterization of private methods, and the combination of 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 [13] 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 [23] 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 [19] 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 [12], for instance, discuss the unsoundness of width subtyping in the presence of object extension. Their solution is to distinguish the types of objects which 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 [17, 18] 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 [15] and Rémy [22] use types that keep track of which methods must be "absent" from an object. Bono, Bugliesi, Dezani, and Liquori [4, 3, 2] 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).

4.3 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. Whether we have chosen the best set of primitives to describe this behavior is, however, open to debate.

Given the choice of primitives, it appears that 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. We are also interested in adapting this work to a setting in which classes are a primitive notion. 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. A much more difficult problem would be to extend our characterization of private data to systems in which self types are allowed to appear contravariantly.

Because of the relationship with implementation strategies for object-oriented languages, we expect the calculus (or some variant thereof) should be implementable in a fairly direct fashion. Nevertheless, as we discussed in Section 4.1, there are still a number of tradeoffs to be explored when implementing the full calculus.

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.

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

References

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

[2] V. Bono, M. Bugliesi, M. Dezani, and L. Liquori. Subtyping constraints for incomplete objects. In *CAAP*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

[3] V. Bono, M. Bugliesi, and L. Liquori. A lambda calculus of incomplete objects. In *Proceedings, Mathematical Foundations of Computer Science*, volume 1113 of *Lecture Notes in Computer Science*, pages 218–229. Springer-Verlag, 1996.

[4] V. Bono and L. Liquori. A subtyping for the Fisher-Honsell-Mitchell calculus of objects. In *Proceedings, Computer Science Logic 1994*, volume 933 of *Lecture Notes in Computer Science*, pages 16–30. Springer-Verlag, 1995.

[5] G. Bracha and G. Lindstrom. Modularity meets inheritance. In *Proceedings of the IEEE Computer Society International Conference on Computer Languages*, pages 282–290, Washington, DC, April 1992. IEEE Computer Society.

[6] K. B. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G. Leavens, and B. C. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):217–238, 1995.

[7] L. Cardelli and P. Wegner. On understanding types, data abstraction and parametric polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[8] M. Felleisen. The theory and practice of first-class prompts. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 180–190. ACM, 1988.

[9] K. Fisher. *Type Systems for Object-Oriented Programming Languages*. PhD thesis, Department of Computer Science, Stanford University, 1996.

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

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

[12] K. Fisher and J. C. Mitchell. The development of type systems for object-oriented languages. *Theory and Practice of Object Systems*, 1:189–220, 1996. Preliminary version appeared in *Proc. Theoretical Aspects of Computer Software,* Springer LNCS 789, 1994, 844–885.

[13] K. Fisher and J. C. Mitchell. 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, 1997. To appear. Revised version to appear in *Theory and Practice of Object Systems*.

[14] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[15] L. Jategaonkar and J. C. Mitchell. Type inference with extended pattern matching and subtypes. *Fundamenta Informaticae*, 19:127–166, 1993. Preliminary version appeared in the *Proceedings of the ACM Symposium on Lisp and Functional Programming*, 1988.

[16] J. Lakos. *Large-Scale C++ Software Design*. Addison-Wesley, 1996.

[17] L. Liquori. An extended theory of primitive objects: First and second order systems. Technical Report CS-23-96, Dipartimento di Informatica, Università di Torino, 1996.

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

[19] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[20] J. C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 109–124. ACM, 1990.

[21] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus Univ., Computer Science Dept., Denmark, 1981.

[22] D. Rémy. Type inference for records in a natural extension of ML. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*, pages 67–95. MIT Press, 1994. An earlier version appeared in the *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, 1989.

[23] D. Rémy and J. Vouillon. Objective ML: A simple object-oriented extension of ML. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 40–53. ACM Press, 1997.

[24] M. Wand. Complete type inference for simple objects. In *Proceedings, Symposium on Logic in Computer Science*, pages 37–44. IEEE, 1987.

A Static Semantics of First-Order System

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

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

$$\frac{\Gamma \vdash \tau}{\Gamma, x{:}\tau \vdash \diamond} \tag{2}$$

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

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash c : \mathsf{typeof}(c)} \tag{3}$$

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash x : \Gamma(x)} \tag{4}$$

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash (\lambda x{:}\tau.e) : (\tau \to \tau')} \tag{5}$$

$$\frac{\Gamma \vdash e : (\tau \to \tau') \qquad \Gamma \vdash e' : \tau}{\Gamma \vdash (e\, e') : \tau'} \tag{6}$$

$$\frac{\Gamma \vdash \tau \preceq \tau' \qquad \Gamma \vdash e : \tau}{\Gamma \vdash e : \tau'} \tag{7}$$

$$\frac{\Gamma \vdash e : \{\!| l : \tau |\!\}}{\Gamma \vdash e.l : \tau} \tag{8}$$

$$\frac{\Gamma \vdash e : \{\!| l : \tau_l{}^{l \in I} |\!\} \qquad \mathsf{Range}(\varphi) \subseteq \mathsf{I}}{\Gamma \vdash e@\varphi : \{\!| l : \tau_{\varphi(l)}{}^{l \in \mathsf{Dom}(\varphi)} |\!\}} \tag{9}$$

$$\frac{\begin{array}{c}\mathsf{Range}(\varphi) \subseteq 1..\mathsf{n} \\ \forall i \in 1..n : \quad \Gamma, s{:}\{\!| j : \tau_j{}^{j \in 1..n} |\!\} \vdash e_i : \tau_i\end{array}}{\Gamma \vdash \mathsf{obj}\, s.\{\!| i \triangleright e_i : \tau_i{}^{i \in 1..n} |\!\}_\varphi : \{\!| l : \tau_{\varphi(l)}{}^{l \in \mathsf{Dom}(\varphi)} |\!\}} \tag{10}$$

$$\frac{\begin{array}{c} m \in I \\ \Gamma \vdash e : \{\!| l : \tau_l{}^{l \in I} |\!\} \qquad \Gamma, s{:}\{\!| l : \tau_l{}^{l \in I} |\!\} \vdash e'_m : \tau_m \end{array}}{\Gamma \vdash e \leftarrow m(s) {=} e'_m : \{\!| l : \tau_l{}^{l \in I} |\!\}} \tag{11}$$

$$\frac{\begin{array}{c} m \notin I \\ \Gamma \vdash e : \{\!| l : \tau_l{}^{l \in I} |\!\} \qquad \Gamma, s{:}\{\!| l : \tau_l{}^{l \in I}, m : \tau'_m |\!\} \vdash e'_m : \tau'_m \end{array}}{\Gamma \vdash e \leftarrow\!\!+ m(s) {=} e'_m : \tau'_m : \{\!| l : \tau_l{}^{l \in I}, m : \tau'_m |\!\}} \tag{12}$$

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

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash b} \tag{13}$$

$$\frac{\Gamma \vdash \tau \qquad \Gamma \vdash \tau'}{\Gamma \vdash \tau \to \tau'} \tag{14}$$

$$\frac{\forall l \in I : \quad \Gamma \vdash \tau_l}{\Gamma \vdash \{\!| l : \tau_l{}^{l \in I} |\!\}} \tag{15}$$

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

$$\frac{\Gamma \vdash \tau}{\Gamma \vdash \tau \preceq \tau} \tag{16}$$

$$\frac{\Gamma \vdash \tau_1 \preceq \tau_2 \qquad \Gamma \vdash \tau_2 \preceq \tau_3}{\Gamma \vdash \tau_1 \preceq \tau_3} \tag{17}$$

$$\frac{\Gamma \vdash \tau'_1 \preceq \tau_1 \qquad \Gamma \vdash \tau_2 \preceq \tau'_2}{\Gamma \vdash \tau_1 \to \tau_2 \preceq \tau'_1 \to \tau'_2} \tag{18}$$

$$\frac{\Gamma \vdash \{\!| l : \tau_l{}^{l \in I \cup J} |\!\}}{\Gamma \vdash \{\!| l : \tau_l{}^{l \in I \cup J} |\!\} \preceq \{\!| l : \tau_l{}^{l \in I} |\!\}} \tag{19}$$

B Static Semantics of Second-Order System

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

$$\frac{}{\bullet\vdash\diamond}\tag{20}$$

$$\frac{\Gamma\vdash\tau \qquad x\notin \mathsf{BV}(\Gamma)}{\Gamma,x{:}\tau\vdash\diamond}\tag{21}$$

$$\frac{\Gamma\vdash\tau \qquad \alpha\notin \mathsf{BV}(\Gamma)}{\Gamma,\alpha\preceq\tau\vdash\diamond}\tag{22}$$

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

$$\frac{\Gamma\vdash\diamond}{\Gamma\vdash c:\mathsf{typeof}(c)}\tag{23}$$

$$\frac{\Gamma\vdash\diamond}{\Gamma\vdash x:\Gamma(x)}\tag{24}$$

$$\frac{\Gamma,x{:}\tau\vdash e:\tau'}{\Gamma\vdash\lambda x{:}\tau.e:\tau\to\tau'}\tag{25}$$

$$\frac{\Gamma\vdash e_1:\tau_2\to\tau \qquad \Gamma\vdash e_2:\tau_2}{\Gamma\vdash e_1\,e_2:\tau}\tag{26}$$

$$\frac{\Gamma\vdash e:\tau' \qquad \Gamma\vdash\tau'\preceq\mathsf{Obj}\,\alpha.\{|l:\tau|\}}{\Gamma\vdash e.l:[\alpha\mapsto\tau']\tau}\tag{27}$$

$$\frac{\Gamma\vdash e:\tau' \qquad \Gamma\vdash v:\tau'\Rightarrow\mathsf{Obj}\,\alpha.\{|l:\tau|\}}{\Gamma\vdash e._v l:[\alpha\mapsto\tau']\tau}\tag{28}$$

$$\frac{\Gamma\vdash e:\tau' \qquad \Gamma\vdash v:\tau'\Rightarrow\tau}{\Gamma\vdash e\&v:\tau}\tag{29}$$

$$\frac{\begin{array}{c}\Gamma\vdash\diamond \qquad \mathsf{Range}(\varphi)\subseteq 1..\mathsf{n}\\ \forall i\in 1..n: \quad \Gamma,\alpha\preceq\mathsf{Obj}\,\alpha.\{|j:\tau_j{}^{j\in 1..n}|\}_{[]},s{:}\alpha\vdash e_i:\tau_i\end{array}}{\Gamma\vdash\mathsf{obj}\,(s{:}\alpha).\{|i\triangleright e_i:\tau_i{}^{i\in 1..n}|\}_\varphi:\mathsf{Obj}\,\alpha.\{|i:\tau_i{}^{i\in 1..n}|\}_\varphi}\tag{30}$$

$$\frac{\begin{array}{c}\Gamma\vdash e_1:\tau \qquad \Gamma\vdash\tau\preceq\mathsf{Obj}\,\alpha.\{|m:\tau_m|\}\\ \Gamma,\alpha\preceq\top_{obj},s{:}\alpha,d{:}\alpha\Rightarrow\tau\vdash e_2:\tau_m\end{array}}{\Gamma\vdash e_1\leftarrow m(s{:}\alpha,d){=}e_2:\tau}\tag{31}$$

$$\frac{\begin{array}{c}\Gamma\vdash e_1:\tau \qquad \Gamma\vdash v:\tau\Rightarrow\mathsf{Obj}\,\alpha.\{|m:\tau_m|\}\\ \Gamma,\alpha\preceq\top_{obj},s{:}\alpha,d{:}\alpha\Rightarrow\tau\vdash e_2:\tau_m\end{array}}{\Gamma\vdash e_1\leftarrow_v m(s{:}\alpha,d){=}e_2:\tau}\tag{32}$$

$$\frac{\begin{array}{c}m\notin I\\ \Gamma\vdash e_1:\mathsf{Obj}\,\alpha.\{|l:\tau_l{}^{l\in I}|\}\\ \Gamma,\alpha\preceq\top_{obj},s{:}\alpha,d{:}\alpha\Rightarrow\mathsf{Obj}\,\alpha.\{|l:\tau_l{}^{l\in I},m:\tau|\}\vdash e_2:\tau\end{array}}{\Gamma\vdash(e_1\leftarrow\!\!+\,l(s{:}\alpha,d){=}e_2:\tau):\mathsf{Obj}\,\alpha.\{|l:\tau_l{}^{l\in I},m:\tau|\}}\tag{33}$$

$$\frac{\Gamma\vdash\mathsf{Obj}\,\alpha.\{|i:\tau_i{}^{i\in 1..n}|\}_\varphi}{\Gamma\vdash\varphi:\mathsf{Obj}\,\alpha.\{|i:\tau_i{}^{i\in 1..n}|\}_{[]}\Rightarrow\mathsf{Obj}\,\alpha.\{|l:i:\tau_i{}^{i\in 1..n}|\}_\varphi}\tag{34}$$

$$\frac{\Gamma\vdash e:\tau' \qquad \Gamma\vdash\tau'\preceq\tau}{\Gamma\vdash e:\tau'}\tag{35}$$

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

$$\frac{\Gamma\vdash\diamond}{\Gamma\vdash b}\tag{36}$$

$$\frac{\Gamma\vdash\diamond \qquad \Gamma=\Gamma',\alpha\preceq\tau,\Gamma''}{\Gamma\vdash\alpha}\tag{37}$$

$$\frac{\Gamma\vdash\tau_1 \qquad \Gamma\vdash\tau_2}{\Gamma\vdash\tau_1\to\tau_2}\tag{38}$$

$$\frac{\Gamma\vdash\tau_1 \qquad \Gamma\vdash\tau_2}{\Gamma\vdash\tau_1\Rightarrow\tau_2}\tag{39}$$

$$\frac{\begin{array}{c}\forall i\in 1..n: \begin{cases}\Gamma,\alpha\preceq\top_{obj}\vdash\tau_i\\ \alpha\text{ occurs covariantly in }\tau_i\end{cases}\\ \Gamma\vdash\diamond \qquad l_1,\ldots,l_n\text{ distinct}\end{array}}{\Gamma\vdash\mathsf{Obj}\,\alpha.\{|l_i:\tau_i{}^{i\in 1..n}|\}}\tag{40}$$

$$\frac{\begin{array}{c}\forall i\in 1..n: \begin{cases}\Gamma,\alpha\preceq\top_{obj}\vdash\tau_i\\ \alpha\text{ occurs covariantly in }\tau_i\end{cases}\\ \Gamma\vdash\diamond \qquad \mathsf{Range}(\varphi)\subseteq 1..\mathsf{n}\end{array}}{\Gamma\vdash\mathsf{Obj}\,\alpha.\{|i:\tau_i{}^{i\in 1..n}|\}_\varphi}\tag{41}$$

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

$$\frac{\Gamma\vdash\tau}{\Gamma\vdash\tau\preceq\tau}\tag{42}$$

$$\frac{\Gamma\vdash\diamond \qquad \Gamma=\Gamma',\alpha\preceq\tau',\Gamma'' \qquad \Gamma\vdash\tau'\preceq\tau}{\Gamma\vdash\alpha\preceq\tau}\tag{43}$$

$$\frac{\Gamma\vdash\tau_1'\preceq\tau_1 \qquad \Gamma\vdash\tau_2\preceq\tau_2'}{\Gamma\vdash\tau_1\to\tau_2\preceq\tau_1'\to\tau_2'}\tag{44}$$

$$\frac{\Gamma\vdash\tau_1'\preceq\tau_1 \qquad \Gamma\vdash\tau_2\preceq\tau_2'}{\Gamma\vdash\tau_1\Rightarrow\tau_2\preceq\tau_1'\Rightarrow\tau_2'}\tag{45}$$

$$\frac{\Gamma\vdash\mathsf{Obj}\,\alpha.\{|l:\tau_l{}^{l\in I\cup J}|\}}{\Gamma\vdash\mathsf{Obj}\,\alpha.\{|l:\tau_l{}^{l\in I\cup J}|\}\preceq\mathsf{Obj}\,\alpha.\{|l:\tau_l{}^{l\in I}|\}}\tag{46}$$

$$\frac{\begin{array}{c}\varphi'\subseteq\varphi\\ \Gamma\vdash\mathsf{Obj}\,\alpha.\{|i:\tau_i{}^{i\in 1..n+m}|\}_\varphi \qquad \mathsf{Range}(\varphi')\subseteq 1..\mathsf{n}\end{array}}{\Gamma\vdash\mathsf{Obj}\,\alpha.\{|i:\tau_i{}^{i\in 1..n+m}|\}_\varphi\preceq\mathsf{Obj}\,\alpha.\{|i:\tau_i{}^{i\in 1..n}|\}_{\varphi'}}\tag{47}$$

$$\frac{\Gamma\vdash\mathsf{Obj}\,\alpha.\{|i:\tau_i{}^{i\in 1..n}|\}_\varphi \qquad I\subseteq\mathsf{Dom}(\varphi)}{\Gamma\vdash\mathsf{Obj}\,\alpha.\{|i:\tau_i{}^{i\in 1..n}|\}_\varphi\preceq\mathsf{Obj}\,\alpha.\{|l:\tau_{\varphi(l)}{}^{l\in I}|\}}\tag{48}$$