Classes = Objects + Data Abstraction

Kathleen Fisher* and John C. Mitchell†
Computer Science Department, Stanford University, Stanford, CA 94305
{kfisher,mitchell}@cs.stanford.edu

January 11, 1996

Abstract

We describe a type-theoretic foundation for object systems that include "interface types" and "implementation types," in the process accounting for access controls such as C++ `private`, `protected` and `public` levels of visibility. Our approach begins with a basic object calculus that provides a notion of object, method lookup, and object extension (an object-based form of inheritance). In this calculus, the type of an object gives an interface, as a set of methods (public member functions) and their types, but does not imply any implementation properties such as the presence or layout of any hidden internal data. We extend the core object calculus with a higher-order form of data abstraction mechanism that allows us to declare supertypes of an abstract type and a list of methods guaranteed not to be present. This results in a flexible framework for studying and improving practical programming languages where the type of an object gives certain implementation guarantees, such as would be needed to statically determine the offset of a function in a method lookup table or safely implement binary operations without exposing the internal representation of objects. We prove type soundness for the entire language using operational semantics and an analysis of typing derivations. Two insights that are immediate consequences of our analysis are the identification of an anomaly associated with C++ `private virtual` functions and a principled, type-theoretic explanation (for the first time, as far as we know) of the link between subtyping and inheritance in C++, Eiffel and related languages.

1 Introduction

In theoretical studies of object systems, such as [AC94b, Bru93, FHM94, PT94] and the earlier papers appearing in [GM94], types are viewed as *interfaces* to objects. This means that the type of an object lists the operations on the object, generally as method names and return types, but does not restrict its implementation. As a result, objects of the same type may have arbitrarily different internal representations. In contrast, the type of an object in common practical object-oriented languages such as Eiffel [Mey92] and C++ [Str86, ES90] may impose some implementation constraints. In particular, although the "private" internal data of an object is not accessible outside the member functions of the class, all objects of the same class must have all of the private internal data listed in the class declaration. In this paper, we present a type-theoretic framework that incorporates both forms of type. We explain the basic principles by extending a core object calculus with a standard higher-order data abstraction mechanism as in [MP88, CW85]. We also

*Supported in part by a Fannie and John Hertz Foundation Fellowship and NSF Grant CCR-9303099.
†Supported in part by NSF Grant CCR-9303099 and the TRW Foundation.

discuss a special-purpose syntax that is closer to common practice and that eliminates a few minor syntactic inconveniences in our specific use of standard abstract data type declarations.

From a programming point of view, "interface" types are often more flexible than types that constrain the implementation of objects. With this form of type, we could define a single type of matrix objects, for example, then represent dense matrices with one form of object and sparse matrices with another. If the type only gives the interface of an object, then both matrix representations could have the same type and therefore be used interchangeably in any program. This kind of flexibility is particularly useful when we write library operations on matrices without assuming any particular implementation. Such library functions may be written using a standard interface type, without concern for how matrices might be implemented in later (or earlier) development of a software system.

Types that restrict the implementations of objects are also important. If we know that all point objects inherit a specific representation of `x` and `y` coordinates, for example, then a program may be optimized to take advantage of this static guarantee. The usual implementations of C++, for example, use type information to statically calculate the offset of member data relative to the starting address of the object. (A similar calculation is used to find the offset of virtual member functions in the v-table at compile time; see [ES90, Section 10.7c].) Such optimizations are not possible in an untyped language such as Smalltalk [GR83] and would not be possible in a typed language where objects of a single type could have arbitrarily dissimilar implementations.

A second, more methodological reason that programmers may be interested in implementation types is that there are greater guarantees of behavioral similarity across subtype hierarchies. More specifically, traditional type systems generally give useful information about the signature (or domain and range) of operations. This is a very weak form of specification and, in many programming situations, it is desirable to have more detailed guarantees. While behavioral specifications are difficult to manipulate effectively, we have a crude but useful approximation when types fix part of the implementation of an object. To return to points, for example, if we know that all subtypes of point share a common implementation of a basic function like `move`, then the type system, in effect, guarantees a behavioral property of points. (This may be achieved in our framework if `move` is private or if we add the straightforward capability of restricting redefinition of protected or public methods.)

A more subtle reason to use types that restrict the implementations of objects has to do with the implementation of binary operations. In an object-oriented context, a binary operation on type A is realized as a member function that requires another A object as a parameter. In a language where all objects of type A share some common representation, it is possible for an A member function to safely access part of the private internal representation of another A object. A simple example of this arises with *set* objects that have only a membership test and a union operation in their public interfaces. With interface types, some objects of type *set* might be represented internally using bit-vectors, while others might use linked lists. In this case, there is no type-safe way to implement union, since no single operation will access both a bit-vector and a linked list correctly. With only interface types, it is necessary to extend the public interface of both kinds of sets to make this operation possible. In contrast, if the type of an object conveys implementation information, then a less flexible but type-safe implementation of set union is possible. In this case, all *set* objects would have one representation and a union operation could be implemented by taking advantage of this uniformity.

This paper presents a provably sound type system, based on accepted type-theoretic principles, that allows us to write both "interface types" and "implementation types". The system is relatively simple in outline, since it may be viewed as a straightforward combination of basic constructs that have been studied previously. However, there are a number of details involving subtype assertions about abstract types, extensions to abstract types, covariance and contravariance of methods, and the absence of methods that make the exact details of the system relatively subtle. In summary, the paper has three main points: (i) the general view that classes correspond to abstract data types, (ii) a precise formulation of a higher-order abstract type mechanism and a flexible underlying object calculus that together make it possible to establish a correspondence between C++-style classes and abstract data types whose representations are objects, (iii) a proof of type soundness for the type system that arises from this analysis. While there is a folkloric belief that C++ and Eiffel classes provide a form of data abstraction, we believe that this is the first technical account suggesting a precise correspondence between class constructs and a standard non-object-oriented form of data abstraction.

Our presentation of classes as abstract data types requires a number of operations on objects. Specifically, the underlying object calculus without data abstraction must provide a basic form of object that allow us to invoke methods of an object, extend objects with new methods, and replace existing methods. Moreover, in order to capture the form of subtyping present in typed object-oriented languages, we must have at least the usual form of subtyping between object types. This much was already provided by our previous object calculus, presented in [FHM94, FM95]; similar primitives are also provided in alternative approaches such as [AC94b, AC94a].

In adding a data abstraction mechanism, we must incorporate subtype specifications, negative information (absence of methods), and variance information in abstract type declarations. This is so that abstract types are extensible in essentially the same way as their underlying representations. Therefore, we extend our basic calculus from [FHM94, FM95] with more detailed static information about type variables and (since most operations are done on functions from types to so-called rows) row variables. While the intuitive decomposition of classes into data abstraction and object primitives is essentially straightforward, the need to maintain detailed information about subtyping properties and extensibility of abstract types leads to certain technical complications and subtleties that had to be overcome in developing this analysis.

A detail for readers of [FHM94] is that the "class" types of that paper were interface types, not classes in the sense of a form of object type that includes implementation information. We therefore renamed them "pro" (for prototype) types in [FM95] and continue that terminology here.

2 Overview by Example

2.1 Protection levels

Each class in an object-oriented program has two kinds of external clients: sections of the program that use objects created from the class and classes that derive new classes from the original. Since the methods of a class may refer to each other, the class also has an internal "client," namely itself. We therefore associate three interfaces with each class. Using C++ terminology, these may be distinguished as follows:

Private methods are only accessible within the implementation of the class,

Protected methods are only accessible in the implementation of the class and derived classes,

Public methods may be accessible, through objects of the class, in any module of the program.

One goal of this paper is to show how we can associate a different type with each interface and use essentially standard type-theoretic constructs to restrict visibility in each part of a program appropriately. In doing so, we pay particular attention to the fact that although the private or protected methods may not be accessible in certain contexts, it is important for the type system to guarantee their existence.

2.2 Point and color point classes

Using C++-like syntax, we might declare classes of points and colored points as shown below. For simplicity, we use only one-dimensional points.

```
 class Point
  private      x   : int;
  protected  setX  : int -> Point;
  public     getX  : int;
               mv  : int -> Point;
         newPoint  : int -> Point;
  end;

class ColorPoint : Point
  private       c  : color;
  protected  setC  : color -> ColorPoint;
  public     getC  : color;
    newColorPoint  : color -> int -> ColorPoint
 end;
```

Intuitively, the `set_` methods are used to assign to private `x`-coordinate or `color` data, and `get_` methods are used to read the values of the data. The move method `mv` changes the `x`-coordinate of a point or colored point. These classes reflect a common idiom of C++ programming, where the basic data fields are kept private so that the class implementor may change the internal representation without invalidating client code. Protected methods make it possible for derived classes to change the values of private data, without providing the same capability outside of derived classes.

In C++ and in the pseudo-code above, each class contains a special function called a *constructor* for the class. Here the constructor is distinguished by the syntactic form `newClassname`. Since all objects of a class are created by calling a class constructor, it is important to be able to call the constructor function before any objects have been created. Therefore, unlike the other functions listed in the class declarations, the class constructor is not a method; it does not belong to objects of the class. Constructors are included in class declarations primarily as a syntactic convenience.

2.3 Interface type expressions

In translating the pseudo-code above into a more precise form, we write a type expression for each interface of each class, resulting in six distinct but related types. In hopes that this practice will provide useful mnemonics, we follow a systematic naming convention where, for class `A`, we call the type expressions for the public, protected, and private interfaces $\mathtt{A}_{\mathtt{pub}}, \mathtt{A}_{\mathtt{prot}}$, and $\mathtt{A}_{\mathtt{priv}}$, respectively.

Although each interface is essentially a list of methods names and their types, it is necessary to use a type function instead of a type for each interface. The reason is that the type associated with the objects instantiated from a given class is recursively defined; this type is a fixed-point of a type function. Point-wise subtyping between such type functions is the critical relation between interfaces for type-checking inheritance.

For Point, this methodology gives us the following type functions, using the form $\langle\!\langle \ldots \rangle\!\rangle$ for object interface types:

$$\begin{aligned}
\texttt{Point}_{\texttt{pub}} &\stackrel{def}{=} \lambda\,\texttt{t}.\langle\!\langle \texttt{getX} : \texttt{int}, \texttt{mv} : \texttt{int} \to \texttt{t} \rangle\!\rangle \\
\texttt{Point}_{\texttt{prot}} &\stackrel{def}{=} \lambda\,\texttt{t}.\langle\!\langle \texttt{setX} : \texttt{int} \to \texttt{t}, \texttt{getX} : \texttt{int}, \texttt{mv} : \texttt{int} \to \texttt{t} \rangle\!\rangle \\
\texttt{Point}_{\texttt{priv}} &\stackrel{def}{=} \lambda\,\texttt{t}.\langle\!\langle \texttt{x} : \texttt{int}, \texttt{setX} : \texttt{int} \to \texttt{t}, \texttt{getX} : \texttt{int}, \texttt{mv} : \texttt{int} \to \texttt{t} \rangle\!\rangle
\end{aligned}$$

These interface functions are formed from the class declaration for Point by replacing occurrences of Point by a type variable t and lambda-abstracting to obtain a type function. We will use *row* variables to range over such type functions, (which map types to finite lists of method/type pairs).

The analogous interfaces for ColorPoint are written using a free row variable p, which will be bound to the abstract type-function for points in the scope where the ColorPoint interfaces will appear:

$$\begin{aligned}
\texttt{ColorPoint}_{\texttt{pub}} &\stackrel{def}{=} \lambda\,\texttt{t}.\langle\!\langle \texttt{p}\,\texttt{t} \mid \texttt{getC} : \texttt{Color} \rangle\!\rangle \\
\texttt{ColorPoint}_{\texttt{prot}} &\stackrel{def}{=} \lambda\,\texttt{t}.\langle\!\langle \texttt{p}\,\texttt{t} \mid \texttt{setC} : \texttt{Color} \to \texttt{t}, \texttt{getC} : \texttt{Color} \rangle\!\rangle \\
\texttt{ColorPoint}_{\texttt{priv}} &\stackrel{def}{=} \lambda\,\texttt{t}.\langle\!\langle \texttt{p}\,\texttt{t} \mid \texttt{c} : \texttt{Color}, \texttt{setC} : \texttt{Color} \to \texttt{t}, \texttt{getC} : \texttt{Color} \rangle\!\rangle
\end{aligned}$$

Since the supertyping bounds on identifier p will give all of the relevant (protected or public) Point methods, there is no need to list the methods inherited from Point. Consequently these ColorPoint interfaces list exactly the same methods as our pseudo-code ColorPoint class. Since the variable p will be "existentially bound" in an abstract data type declaration, the occurrence of p in each expression will guarantee that all the private methods of Point objects are present in every ColorPoint object, without exposing any other information about private Point methods. So-called "kind" information in the declaration of p will guarantee that methods named c, setC, and getC are not present, making it type-safe to extend p objects with these new methods.

2.4 Implementations

A class implementation specifies an object layout and set of method bodies (code for the methods of the objects). In our approach, the object layout will be given by a type expression and the method bodies will be part of the constructor function. Following general principles of data abstraction, the method bodies may rely on aspects of the representation that are hidden from other parts of the program.

We use a subtype-bounded form of data abstraction based on existential types [MP88, CW85]. Using this formalism, the implementation of points will be given by a pair with subtype-bounded existential type of the form

$$\{\!|\texttt{p} <: (\ldots) :: \kappa = \texttt{P}_{\texttt{priv}},\ \ \texttt{ConsImp}_{\texttt{p}}|\!\}$$

consisting of the private interface $\texttt{P}_{\texttt{priv}}$ for points and a constructor function $\texttt{ConsImp}_{\texttt{p}}$ that might use an initial value for the x-coordinate, for example, to return a new point object. (Our framework allows any number of constructor functions, or other "non-virtual" operations to be provided here. However, for simplicity, we discuss only the special case of one constructor per class.) For the moment, we leave the supertype bound, indicated by the ellipsis (...) above, unspecified since we later discuss two separate approaches, a minimal one in which we give the protected interface here (later restricting the program view to the public interface) and a more special-purpose approach in which both the protected and public interfaces are specified in the class implementation. The "kind"

```
Abstype  $\mathtt{p} <:_w \mathtt{P_{prot}} :: \mathtt{T}^+ \rightarrow (\{\mathtt{c}, \mathtt{getC}, \mathtt{setC}\}; \emptyset)$
   with  $\mathtt{newPoint} : \mathtt{Int} \rightarrow \mathbf{pro}\, \mathtt{u.p\, u}$
     is  $\{\!|\mathtt{p} <:_w \mathtt{P_{prot}} :: (\mathtt{T}^+ \rightarrow (\{\mathtt{c}, \mathtt{getC}, \mathtt{setC}\}; \emptyset)) = \mathtt{P_{priv}},\ \mathtt{Imp_p}|\!\}$
     in
         Abstype  $\mathtt{cp} <:_w \mathtt{CP_{prot}} :: \mathtt{T}^+ \rightarrow (\emptyset; \emptyset)$
            with  $\mathtt{newColorPoint} : \mathtt{Int} \rightarrow \mathtt{Color} \rightarrow \mathbf{pro}\, \mathtt{u.cp\, u}$
              is  $\{\!|\mathtt{cp} <:_w \mathtt{CP_{prot}} :: (\mathtt{T}^+ \rightarrow (\emptyset; \emptyset)) = \mathtt{CP_{priv}},\ \mathtt{Imp_{cp}}|\!\}$
              in
                  Abstype  $\mathtt{p} <:_w \mathtt{P_{pub}} :: \mathtt{T}^+ \rightarrow (\emptyset; \emptyset)$
                     with  $\mathtt{newPoint} : \mathtt{Int} \rightarrow \mathbf{obj}\, \mathtt{t.p\, t}$
                       is  $\{\!|\mathtt{p} <:_w \mathtt{P_{pub}} :: (\mathtt{T}^+ \rightarrow (\emptyset; \emptyset)) = \mathtt{p},\ \mathtt{newPoint}|\!\}$
                       in
                           Abstype  $\mathtt{cp} <:_w \mathtt{CP_{pub}} :: \mathtt{T}^+ \rightarrow (\emptyset; \emptyset)$
                              with  $\mathtt{newColorPoint} : \mathtt{Int} \rightarrow \mathtt{Color} \rightarrow \mathbf{obj}\, \mathtt{t.cp\, t}$
                                is  $\{\!|\mathtt{cp} <:_w \mathtt{P_{pub}} :: (\mathtt{T}^+ \rightarrow (\emptyset; \emptyset)) = \mathtt{cp},\ \mathtt{newColorPoint}|\!\}$
                                in
                                    ⟨Program⟩
```

Figure 1: Nested abstract data types for points and colored point classes.

κ will indicate that we are declaring an abstract type function, list methods that are guaranteed not to be present in the implementation of points, and describe the variance of point objects. The variance information is needed to infer subtyping relationships for object types that contain row variable `p`.

The implementation of `ColorPoint` similarly has the form

$$\{\!|\mathtt{cp} <: (\ldots) :: \kappa' = \mathtt{CP_{priv}},\ \mathtt{ConsImp_{cp}}|\!\}$$

where the constructor $\mathtt{ConsImp_{cp}}$, first invokes the `Point` class constructor `newPoint`, then extends the resulting prototype with the new methods `c`, `setc`, and `getc`. Definitions of the `Point` and `ColorPoint` class constructors are given in Section 3.4.

2.5 Class hierarchies as nested abstract types

Our basic view of classes and implementation types is that the class-based pseudo-code in Section 2.2 may be regarded as sugar for the sequence of nested abstract data type (`Abstype`) declarations given in Figure 1. Since it is syntactically awkward to use two declarations per class, one giving the protected interface and the other the public interface, we discuss alternate syntactic presentations in Section 3.5.

In order, the four abstract type declarations give the protected view of `Point`, the protected view of `ColorPoint`, the public view of `Point`, and the public view of `ColorPoint`. The nesting structure allows the implementation of `ColorPoint` to refer to the protected view of `Point` and allows the program to refer to public views of both classes.

The two inner declarations hide the protected view of a class by redeclaring the same type name and constructor and exposing the public view (with a different type, as discussed below). Since the implementation of the public view, in each case, is exactly the same as the implementation of the protected view, hiding the protected methods is the only function of the two inner declarations. We admit that reusing bound variables is a bit of a "hack," but this is a relatively minor issue that can be solved by departing from the simple block-structured scoping mechanism we use here.

One distinction between the protected and public views is that the constructors for the protected view return an object with a **pro** type while the public view constructors return objects with **obj** types. These two types from our underlying object calculus allow different sets of operations on objects. Specifically, new methods may be added to an object with a **pro** type and existing methods can be overridden. If an object has an **obj** type, on the other hand, the only operation is to invoke a method of the object. Since we have different sets of operations, there are different subtyping rules: the subtyping relation is relatively rich between **obj** types, while the only supertypes of a **pro** type are object types. Intuitively, this means that the methods of an object may be modified or extended when it is used as a prototype, but not once it is "promoted" to a member of an **obj** type. While the distinction between these two kinds of types is convenient for our purposes here, it was originally devised as a mechanism for obtaining nontrivial subtyping in the presence of object inheritance primitives (object extension and method redefinition).

3 Object calculus and type system

3.1 The calculus

The expressions of our core calculus are untyped lambda terms, including variable $\mathtt{x}$, application $\mathtt{e}_1\mathtt{e}_2$ and lambda abstraction $\lambda\mathtt{x}.\,\mathtt{e}$, extended with four object forms:

$\langle\rangle$	the empty prototype or object
$\mathtt{e} \Leftarrow \mathtt{m}$	send message $\mathtt{m}$ to prototype or object $\mathtt{e}$
$\langle \mathtt{e}_1 \leftarrow\!\!+\, \mathtt{m}{=}\mathtt{e}_2\rangle$	extend prototype $\mathtt{e}_1$ with new method $\mathtt{m}$ having body $\mathtt{e}_2$
$\langle \mathtt{e}_1 \leftarrow \mathtt{m}{=}\mathtt{e}_2\rangle$	replace prototype $\mathtt{e}_1$'s method body for $\mathtt{m}$ by $\mathtt{e}_2$

These expressions are the same as in [FHM94, FM95]. However, the type system used in the present paper is more detailed. We extend this core calculus with two encapsulation primitives:

$$\mathtt{Abstype}\ \mathtt{r} <:_{\mathtt{w}} \mathtt{R}\ ::\kappa\ \mathtt{with}\ \mathtt{x} : \tau\ \mathtt{is}\ \mathtt{e}_1\ \mathtt{in}\ \mathtt{e}_2$$
$$\{\!|\mathtt{r} <:_{\mathtt{w}} \mathtt{R}\ ::\kappa = \mathtt{R}',\ \mathtt{e}|\!\}$$

The first is used to provide limited access to implementation $\mathtt{e}_1$ in client $\mathtt{e}_2$. Type expression $\mathtt{r} <:_{\mathtt{w}} \mathtt{R}\ ::\kappa$ and assumption $\mathtt{x} : \tau$ provide the interface for this access. The type system will require expression $\mathtt{e}_1$ to have the form $\{\!|\mathtt{r} <:_{\mathtt{w}} \mathtt{R}\ ::\kappa = \mathtt{R}',\ \mathtt{e}|\!\}$, which is the implementation of the abstraction. The components of these expressions will be explained in more detail after we introduce the type system.

3.2 Operational semantics

The operational semantics include β-reduction for evaluating function applications and a $(\Leftarrow)$ rule for evaluating message sends:

$$\langle \mathtt{e}_1 \leftarrow\!\!\circ\, \mathtt{m}{=}\mathtt{e}_2\rangle \Leftarrow \mathtt{m} \xrightarrow{eval} \mathtt{e}_2\,\langle \mathtt{e}_1 \leftarrow\!\!\circ\, \mathtt{m}{=}\mathtt{e}_2\rangle$$

where $\leftarrow\!\!\ominus$ may be either $\longleftarrow\!\!+$ or $\longleftarrow$. We also need various bookkeeping rules to access methods defined within $\texttt{e}_1$. These bookkeeping rules appear in Appendix A; they are explained in full in [FHM94]. The reduction rule (*Abstype*) for abstract data type declarations is:

$$\texttt{Abstype r} <:_{\texttt{w}} \texttt{R} \; :: \kappa \; \texttt{with x} : \tau \; \texttt{is} \; \{\!|\texttt{r} <:_{\texttt{w}} \texttt{R} \; :: \kappa = \texttt{R}', \; \texttt{e}_1|\!\} \; \texttt{in e}_2 \; \xrightarrow{eval} \; [\texttt{R}'/\texttt{r}, \texttt{e}_1/\texttt{x}]\texttt{e}_2$$

3.3 Static Type System

The type expressions, given in Appendix B, include type variables, function types, **pro** types, **obj** types, and existential types. To reduce the complexity of the type system, these types are divided into two categories. The unquantified, monotypes are indicated using metavariables $\tau, \tau', \tau_1, \ldots$. The quantified types, indicated using metavariables $\sigma, \sigma', \sigma_1, \ldots$, may contain existential quantifiers.

A *row* is a finite list of *method name, type* pairs. Row expressions appear as subexpressions of type expressions, with rows and types distinguished by kinds. Intuitively, the elements of kind $(\{\vec{m}\}; V)$ are the rows that do *not* include the method names in $\{\vec{m}\}$ and whose the free type variables appear with variance indicated in variance set V. We keep track of the absence of methods in order to guarantee that methods are not multiply defined. The variance information, which tells whether a variable appears monotonically, antimonotonically, or neither, is necessary for subtyping judgements involving types of the form $\mathbf{pro}\, t.R$ or $\mathbf{obj}\, t.R$ since R may contain row variables. Type functions, which arise as row expressions with kind $T^a \to (\{\vec{m}\}; V)$, are used to infer a form of higher-order polymorphism for method bodies and to provide type interfaces to encapsulated implementations. The annotation a indicates the variance of the abstracted variable.

To avoid unnecessary repetition in our presentation, we use the meta-variable **probj** for either **obj** or **pro**. Intuitively, the elements of type $\mathbf{probj}\, t.\langle\!\langle m_1 : \tau_1, \ldots, m_k : \tau_k\rangle\!\rangle$ are objects e such that for $1 \leq i \leq k$, the result of $e \Leftarrow m_i$ is a value of type τ_i. However, since the bound type variable t may appear free in τ_i, the type of $e \Leftarrow m_i$ is actually the result of replacing each free occurrence of t in τ_i by $\mathbf{probj}\, t.\langle\!\langle m_1 : \tau_1, \ldots, m_k : \tau_k\rangle\!\rangle$. Because of this substitution, $\mathbf{probj}\, t.\langle\!\langle \ldots\rangle\!\rangle$ is effectively a special form of recursively-defined type.

The typing rule for sending a message to an object is

$$(probj \Leftarrow) \qquad \frac{\begin{array}{c} , \vdash e : \mathbf{probj}\, t.R \\ , , t : \{t^+\} \vdash R <:_w \langle\!\langle m : \tau\rangle\!\rangle \end{array}}{, \vdash e \Leftarrow m : [\mathbf{probj}\, t.R/t]\tau}$$

where the substitution for t in τ reflects the recursive nature of **pro** and **obj** types. This rule differs from the ones given in [FHM94, FM95] by requiring only that we may derive $, , t : \{t^+\} \vdash R <:_w \langle\!\langle m : \tau\rangle\!\rangle$ instead of the more stringent requirement that $R \equiv \langle\!\langle R' | m : \tau\rangle\!\rangle$. This relaxation permits us to type message sends to objects whose types may be partially abstract (*i.e.*, types containing row variables). Another thing to notice about this rule is the subscript w on the subtyping judgment. This symbol indicates that only *width* subtyping was used to reach this conclusion. Our system supports both width and depth subtyping on object types; however, for soundness of certain operations, it is essential to keep track of exactly where depth subtyping occurs.

The rule for method override is as follows:

$$(pro\ over) \qquad \frac{\begin{array}{l} , \vdash e_1 : \mathbf{pro}\, u.R \\ , , u : \{u^+\} \vdash R <:_w \langle\!\langle m : \tau\rangle\!\rangle \\ , , I_r \vdash e_2 : [\mathbf{pro}\, u.ru/t](t \to \tau) \end{array}}{, \vdash \langle e_1 \leftarrow m{=}e_2\rangle : \mathbf{pro}\, u.R}$$

$$\text{where } I_r = r <:_w \lambda t.R :: T^0 \rightarrow (\emptyset;\, \mathit{Invar}(V_\Gamma)).$$

The primary difference between this rule and the one given in [FHM94, FM95] is again the relaxation of the requirement that R be of the form $\langle\!\langle R' \mid m : \tau \rangle\!\rangle$, which permits us to override methods in objects with partially abstract types.

The distinction between **pro** and **obj** types is that the former allows redefinition or extension of methods, while the latter gives subtyping instead. An inessential artifact of our calculus is that we may use the same untyped terms for prototypes and objects. We might like to use this coincidence to advantage by using the following rule (along with subsumption) to convert prototypes to objects:

$$(\mathit{seal\text{-}unsound}) \qquad \frac{,\ \vdash \mathbf{obj}\, u.R : T}{,\ \vdash \mathbf{pro}\, u.R <: \mathbf{obj}\, u.R}$$

where the hypothesis $,\ \vdash \mathbf{obj}\, u.R : T$ ensures that $\mathbf{obj}\, u.R$ is well formed. Unfortunately, this rule is problematic when the variable u appears contravariantly in R. Hence we need a more complicated rule that we combine with the rule for **obj** subtyping since they both have the same form:

$$(<:\ \mathit{obj}) \qquad \frac{\begin{array}{c} ,\ ,\ t : \{t^+\} \vdash R_1 <:_B R_2 \\ ,\ ,\ t : \{t^+\} \vdash R_2 :: (M; V) \\ \mathit{Var}(t, V) \in \{?, +\} \end{array}}{,\ \vdash \mathbf{probj}\, t.R_1 <: \mathbf{obj}\, t.R_2}$$

Because R_1 and R_2 in this rule might contain row variables, we cannot use their syntactic form to determine whether the critical type variable appears covariantly (monotonically). Therefore, we appeal to the kind of R_2, which contains so-called variance (monotonicity or antimonotonicity) information. The auxiliary function $\mathit{Var}(t, V)$ gives the variance of type variable of t in variance set V. If $\mathit{Var}(t, V)$ is ?, then t does not appear in R_2, and hence is vacuously covariant. If $\mathit{Var}(t, V)$ is $+$, then t appears covariantly. In either case, the consequent of the rule follows from the premises. Since the subscript B on the subtype relation is unconstrained, this rule gives us subtyping in both width and depth for **obj** types . This rule also has the somewhat unfortunate property that we cannot convert prototypes to **obj** type if there are methods that are contravariant in the bound type variable. However, this appears to be a fundamental trade-off and limitation.

The rule for forming a class implementation: is the standard rule for existential introduction, extended to address kinding and supertyping constraints.

$$(\exists <:\ \mathit{intro}) \qquad \frac{\begin{array}{c} ,\ \vdash R_1 :: \kappa \\ ,\ \vdash R_1 <:_w R \\ ,\ \vdash e : [R_1/r]\tau \end{array}}{,\ \vdash \{\!|\, r <:_w R :: \kappa = R_1,\ e \,|\!\} : \exists(r <:_w R :: \kappa)\tau}$$

The rule for existential types is standard as well. The full type system appears in Appendix C.

3.4 Examples

The `Point` class constructor $\mathtt{Imp_P}$ from Section 2.4 may be written as follows:

$$\begin{array}{lrcl} \lambda\,\mathtt{initX}.\langle & \mathtt{x} & = & \lambda\,\mathtt{self}.\mathtt{initX}, \\ & \mathtt{setX} & = & \lambda\,\mathtt{self}.\lambda\,\mathtt{newX}.\langle \mathtt{self} \leftarrow \mathtt{x} = \lambda\,\mathtt{self'}.\mathtt{newX} \rangle, \\ & \mathtt{getX} & = & \lambda\,\mathtt{self}.(\mathtt{self} \Leftarrow \mathtt{x}), \\ & \mathtt{mv} & = & \lambda\,\mathtt{self}.\lambda\,\mathtt{dx}.(\mathtt{self} \Leftarrow \mathtt{setX}(\mathtt{dx} + \mathtt{self} \Leftarrow \mathtt{getX}))\rangle \end{array}$$

The ColorPoint constructor $\mathtt{Imp}_{\mathtt{cp}}$ first invokes the Point class constructor newPoint, (which is implemented by the above $\mathtt{Imp}_{\mathtt{p}}$ function) then extends the resulting prototype with the new methods c, setc, and getc:

$$\begin{array}{llll}
\lambda\,\mathtt{iX}.\lambda\,\mathtt{iC}.\ \langle\langle\langle\langle \mathtt{newPoint(iX)} & \leftarrow\!+ & \mathtt{c} & = \lambda\,\mathtt{self}.\mathtt{iC}\rangle \\
 & \leftarrow\!+ & \mathtt{getC} & = \lambda\,\mathtt{self}.(\mathtt{self} \Leftarrow \mathtt{c})\rangle \\
 & \leftarrow\!+ & \mathtt{setC} & = \lambda\,\mathtt{self}.\lambda\,\mathtt{newC}.\langle \mathtt{self} \leftarrow \mathtt{c} = \lambda\,\mathtt{self}'.\mathtt{newC}\rangle\rangle\rangle
\end{array}$$

By first calling the constructor for the Point class, the ColorPoint class permits the parent Point class to build the parts of the ColorPoint objects inherited from Point, including the private components that the ColorPoint class cannot access.

3.5 An explicit class construct

Although the pattern of nested abstract data type declarations discussed in Section 2.5 gives us some useful insight into classes, this is an "encoding" of a relatively complex construct into a pattern of use of two simpler ones; it not a class construct that could be used as the basis for programming language design. In this brief section, we discuss an alternate "foundational object calculus" with classes taken as basic instead of derived from abstract data type declarations. While additional redesign is possible, we confine ourselves to replacing patterns of abstype by a single class construct; we do not consider modifications of the underlying core object calculus. As a simplifying syntactic assumption, we continue to work with block-structured syntax, instead of a more complicated module formalism with visibility controlled by explicit uses declarations.

A simple construct that may be used in place of standard abstype is a class declaration of the form

```
Class   p <: P_prot <: P_pub :: kind_p with newP(pars_p) is <P_priv, Imp_p>
and     q <: Q_prot <: Q_pub :: kind_q with newQ(pars_q) is <Q_priv, Imp_q>
and     ...
and     r <: R_prot <: R_pub :: kind_r with newR(pars_r) is <R_priv, Imp_r>
in
    Program
end
```

We can derive direct typing rules for this construct using the translation into nested abstype's.

The idea is that classes q . . . r may declare subtypes of p. Therefore, this declaration binds p and newP in the subsequent class declaration clauses and in Program. However, the subtyping information about p and type of newP will be different in the two scopes. Specifically, the clauses giving possible subtypes of p are type-checked using the assumptions $\mathtt{p} <: \mathtt{P\_prot} :: \mathtt{kind\_p}$ and $\mathtt{newP} : \mathtt{pars\_p} \to \mathbf{pro}\,u.\mathtt{p}\,u$, while the program is type-checked using the assumptions $\mathtt{p} <: \mathtt{P\_pub} :: \mathtt{kind\_p}$ and $\mathtt{newP} : \mathtt{pars\_p} \to \mathbf{obj}\,u.\mathtt{p}\,u$. This makes it possible for the implementation of derived classes q . . . r to access the protected methods of p and extend the implementation of p objects by adding new methods or redefining existing ones. Each of the subsequent class declaration clauses are treated similarly so that, for example, the declaration of r has analogous access to the protected methods of q. As in the pattern of abstype declarations given in Section 2.5, the client program does not have these capabilities.

Although we have explained this construct under the assumption that q . . . r are subtypes of p, they are more properly viewed as derived classes that may or may not result in subtypes. In particular, since there is no restriction on the syntactic form of Q_pub, for example, this could be an expression giving **obju.qu** <: **obju.pu**, or it might not. If it is not, then we have the analog of C++ private base classes where inheritance does not produce a subtype.

4 Overview of soundness proof

The soundness proof, using the operational semantics discussed in Section 3.2, has two parts. The first shows that evaluation preserves type; this property is traditionally called subject reduction. The second part established that no typable expression evaluates to *error*. In the second part, we use a specific evaluation strategy, described in [FHM94], to carry out the argument. The two parts together guarantee that we never obtain a "message-not-understood" errors from any typable expression.

While the details are different, the proof follows the same outline as [FHM94, FM95]. A sketch of the proof appears in Appendix D.

Theorem 4.1 (Subject Reduction) *If* $, \vdash e : \tau$ *is derivable, and* $e \xrightarrow{eval} e'$*, then* $, \vdash e' : \tau$ *is also derivable.*

Theorem 4.2 (Type Soundness) *If the judgement* $\epsilon \vdash e : \tau$ *is derivable, then* $eval(e) \neq error$*, where the function eval is as in [FHM94], extended with rules for* $(Class)$ *reduction.*

5 Analysis

Here we briefly note several implications of our approach.

5.1 Representation independence for classes

One advantage of our decomposition of classes into object operations and standard data abstraction is that a number of properties developed in the analysis of traditional data abstraction without objects may be applied to object-oriented languages. For example, the results in [Rey83, MM85, Mit86, Mit91] give various sufficient conditions on interchangeability of implementations. Put briefly, we may replace one implementation of a class with another, in any program, as long as the protect and public interfaces of the new implementation conform to the old ones and the observable behaviors correspond.

5.2 Subtyping and inheritance

A basic issue in the literature on object-oriented programming is the relation between subtyping and inheritance. An early and influential paper, [Sny86], argues that the way these two ideas are often linked (as in Eiffel and C++) is inconvenient and unfounded. We believe that the arguments in [Sny86, Coo92] and related papers are essentially correct for interface types: subtyping between interface types has nothing to do with the way objects are implemented. However, the analysis in the present paper shows that for implementation types, inheritance may be necessary (although not always sufficient) to produce a subtype. In short, if a type t is abstract, in the sense that all or part of its implementation is hidden, then the only safe way to define a subtype of t is by extending the hidden implementation.

5.3 "Private virtual" functions

There are some interesting problems associated with C++ `private virtual` functions. In our representation of classes, these show up immediately in the need to include "negative information" in the interface of an abstract type. This is necessary if if we want to be able to extend the representation, since otherwise we might try to add a new method that replaces an existing method, invalidating another method that depends on the presence of the one that is replaced. (See [FM94], for example, for further discussion.) However, this form of negative information is not used in C++ class interfaces or header files. Therefore, we have an anomalous situation in which private virtual functions, which are not supposed to be visible to derived classes, can be unintentionally overridden because in addition to being "private," they are also declared "virtual," which means they are allowed to be redefined in derived classes.

6 Conclusion

In this paper, we describe a type-theoretic model of various levels of encapsulation and visibility in object-oriented systems. More specifically, we show that classes, of the form found in C++, Eiffel and related languages, may be regarded as the combination of two orthogonal language features: a form of objects without encapsulation and a standard form of data abstraction mechanism (albeit higher-order and including subtype constraints). This intuitive view explains the correlation between subtyping and inheritance and answers some of the criticism found in papers such as [Sny86, Coo92]; other advantages of our approach are outlined in Section 5.

Several other features of C++-like object systems can be modeled in our framework. In particular, we can account for *friend* functions, non-virtual functions and inheritance with private base classes (i.e., inheritance that does not result in a subtype) using the concepts presented here.

There are a number of promising directions for further work. One direction is to examine further features of C++, with the goal of identifying anomalies or simplifying the language. An interesting topic in this vein is the incorporation of "abstract" classes. An issue related to both improving C++-like systems and the general problem of name spaces for method names might be to develop a binding mechanism so that method names remain local to their intended scope. This could resolve the problem with C++ `private virtual` functions and also simplify our type system by eliminating the need for negative kind information in abstract type declarations. With an eye toward future language design, we also hope to refine the special syntax described briefly in Section 3.5 and consider the reformulation of our block-structured constructs using modules and dot notation [CL90, HL94]

References

[AC94a] M. Abadi and L. Cardelli. A theory of primitive objects: second-order systems. In *Proc. European Symposium on Programming*, pages 1–24. Springer-Verlag, 1994.

[AC94b] M. Abadi and L. Cardelli. A theory of primitive objects: untyped and first-order systems. In *Proc. Theor. Aspects of Computer Software*, pages 296–320. Springer-Verlag LNCS 789, 1994.

[Bru93] K. Bruce. Safe type checking in a statically-typed object-oriented programming language. In *Proc 20th ACM Symp. Principles of Programming Languages*, pages 285–298, 1993.

[CL90] Luca Cardelli and Xavier Leroy. Abstract types and the dot notation. Technical Report 56, DEC Systems Research Center, Palo Alto, CA, March 1990.

[Coo92] W.R. Cook. Interfaces and specifications for the Smalltalk-80 collection classes. In *ACM Conf. Object-oriented Programming: Systems, Languages and Applications*, pages 1–15, 1992.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[ES90] M. Ellis and B. Stroustrop. *The Annotated C^{++} Reference Manual.* Addison-Wesley, 1990.

[FHM94] 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.

[FM94] K. Fisher and J.C. Mitchell. Notes on typed object-oriented programming. In *Proc. Theoretical Aspects of Computer Software*, pages 844–885. Springer LNCS 789, 1994.

[FM95] K. Fisher and J. Mitchell. A delegation-based object calculus with subtyping. In *Fundamentals of Computation Theory (FCT'95).* Springer LNCS, 1995. To appear.

[GM94] C.A. Gunter and J.C. Mitchell, editors. *Theoretical aspects of object-oriented programming.* MIT Press, Cambridge, MA, 1994.

[GR83] A. Goldberg and D. Robson. *Smalltalk–80: The language and its implementation.* Addison Wesley, 1983.

[HL94] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proc. 21-st ACM Symp. on Principles of Programming Languages*, 1994.

[Mey92] B. Meyer. *Eiffel: The Language.* Prentice-Hall, 1992.

[Mit86] J.C. Mitchell. Representation independence and data abstraction. In *Proc. 13th ACM Symp. on Principles of Programming Languages*, pages 263–276, January 1986.

[Mit91] J.C. Mitchell. On the equivalence of data representations. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 305–330. Academic Press, 1991.

[MM85] J.C. Mitchell and A.R. Meyer. Second-order logical relations. In *Logics of Programs*, pages 225–236, Berlin, June 1985. Springer-Verlag LNCS 193.

[MP88] J.C. Mitchell and G.D. Plotkin. Abstract types have existential types. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, 1988. Preliminary version appeared in *Proc. 12th ACM Symp. on Principles of Programming Languages,* 1985.

[PT94] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming,* 4(2):207–248, 1994.

[Rey83] J.C. Reynolds. Types, abstraction, and parametric polymorphism. In *Information Processing '83*, pages 513–523. North-Holland, Amsterdam, 1983.

[Sny86] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proc. ACM Symp. on Object-Oriented Programming Systems, Languages, and Applications*, pages 38–46, October 1986.

[Str86] B. Stroustrop. *The C^{++} Programming Language.* Addison-Wesley, 1986.

A Syntax and operational semantics of expressions

Expressions

$$\begin{array}{l}
e \ ::= \ x \mid c \mid \lambda x.\, e \mid e_1 e_2 \mid \\
\langle\rangle \mid e \Leftarrow m \mid \langle e_1 \leftarrow m = e_2 \rangle \mid \langle e_1 \leftarrow\!\!+\ m = e_2 \rangle \mid \\
\{\!| r <:_w R_1 \ ::\kappa = R_2,\ e |\!\} \mid \\
\mathit{Abstype}\ r <:_w R \ ::\kappa\ \mathit{with}\ x : \rho\ \mathit{is}\ e_1\ \mathit{in}\ e_2
\end{array}$$

Operational Semantics:

$(\mathit{switch\ ext\ ov})$	$\langle\langle e_1 \leftarrow m_2{=}e_2\rangle \leftarrow\!\!+\ m_3{=}e_3\rangle$	$\xrightarrow{book}$	$\langle\langle e_1 \leftarrow\!\!+\ m_3{=}e_3\rangle \leftarrow m_2{=}e_2\rangle$
$(\mathit{perm\ ov\ ov})$	$\langle\langle e_1 \leftarrow m_2{=}e_2\rangle \leftarrow m_3{=}e_3\rangle$	$\xrightarrow{book}$	$\langle\langle e_1 \leftarrow m_3{=}e_3\rangle \leftarrow m_2{=}e_2\rangle$
$(\mathit{add\ ov})$	$\langle e_1 \leftarrow\!\!+\ m_2{=}e_2\rangle$	$\xrightarrow{book}$	$\langle\langle e_1 \leftarrow\!\!+\ m_2{=}e_2\rangle \leftarrow m_2{=}e_2\rangle$
$(\mathit{cancel\ ov\ ov})$	$\langle\langle e_1 \leftarrow m_3{=}e_2\rangle \leftarrow m_3{=}e_3\rangle$	$\xrightarrow{book}$	$\langle e_1 \leftarrow m_3{=}e_3\rangle$
(β)	$(\lambda x.\, e_1)e_2$	$\xrightarrow{eval}$	$[e_2/x]e_1$
$(\Leftarrow)$	$\langle e_1 \leftarrow\!\!\circ\ m{=}e_2\rangle \Leftarrow m$	$\xrightarrow{eval}$	$e_2\langle e_1 \leftarrow\!\!\circ\ m{=}e_2\rangle$ where $\leftarrow\!\!\circ$ may either $\leftarrow\!\!+$ or $\leftarrow$.
$(\mathit{Abstype})$	$\mathit{Abstype}\ r <:_w R\ ::\kappa\ \mathit{with}\ x:\tau$ $\mathit{is}\ \{\!| r <:_w R\ ::\kappa = R',\ e_1 |\!\}\ \mathit{in}\ e_2$	$\xrightarrow{eval}$	$[R'/r, e_1/x]e_2$

B Type System

Types

$$\begin{array}{l}
\sigma \ ::= \ \tau \mid \exists(r <:_w R\ ::\kappa)\tau \\
\tau \ ::= \ t \mid \tau_1 \to \tau_2 \mid \mathbf{pro}\,t.R \mid \mathbf{obj}\,t.R
\end{array}$$

Rows

$$R \ ::= \ r \mid \langle\!\langle\rangle\!\rangle \mid \langle\!\langle R \mid m : \tau \rangle\!\rangle \mid \lambda t.\, R \mid R\tau$$

Kinds

$$\begin{array}{ll}
\mathit{kind} \ ::= \ V \mid \kappa & \\
V \ ::= \ \{\vec{t^b}\}, & b \ ::= \ + \mid - \mid o \\
\kappa \ ::= \ T^a \to \nu \mid \nu, & a \ ::= \ b \mid ? \\
\nu \ ::= (M; V) & \\
M \ ::= \ \{\vec{m}\} &
\end{array}$$

Contexts

$$, \ ::= \ \epsilon \mid ,\ ,x:\tau \mid ,\ ,t:V \mid ,\ ,r <:_w R\ ::\kappa$$

The judgement forms are:

$, \vdash *$	well-formed context
$, \vdash e : \tau$	term has type
$, \vdash \sigma : V$	well-formed type with variance V
$, \vdash R_1 <:_B R_2$	row R_1 subtype of R_2 B gives width vs. depth
$, \vdash \tau_1 <: \tau_2$	type τ_1 subtype of τ_2
$, \vdash R :: \kappa$	row has kind

The subtyping annotations are:

Width vs. Depth

$B ::= w \mid d \mid w, d \mid B_1 + B_2$ Indicate record subtyping forms.
"+" is the union of B_1 and B_2.

Operations on Variance Sets:

$$\begin{array}{lcl}
V_1 - V_2 & = & \text{set difference} \\
\overline{\{t_1^{a_1}, \ldots, t_n^{a_n}\}} & = & \{t_1^{\overline{a_1}}, \ldots, t_n^{\overline{a_n}}\}, \text{ where } \overline{a_i} \text{ inverts the sign of } a_i. \\
D(V_1, \ldots, V_n) & = & \{t \mid t^a \in V_i \text{ for some } a, i\} \\
Var(t, V) & = & \begin{cases} a & \text{if } t^a \in V \\ ? & \text{otherwise} \end{cases} \\
V \backslash t & = & V - \{t^{Var(t,V)}\} \\
GVar(t, V_1, \ldots, V_n) & = & lub\{Var(t, V_1), \ldots, Var(t, V_n)\} \\
Merge(V_1, \ldots, V_n) & = & \{t^{GVar(t, V_1, \ldots, V_n)} \mid t \in D(V_1, \ldots, V_n)\} \\
Invar(V) & = & \{t^o \mid t \in D(V)\} \\
V_\Gamma & = & \{t^a \mid t^a \in dom(,)\}
\end{array}$$

where lub is taken with respect to the ordering: $+ \leq o$, $- \leq o$, $? \leq -$, $? \leq +$.

Variance Substitutions

$$[V_2/t]V_1 = \begin{cases} Merge(V_1', V_2) & \text{if } V_1 = V_1', t^+ \\ Merge(V_1', \overline{V_2}) & \text{if } V_1 = V_1', t^- \\ Merge(V_1', Invar(V_2)) & \text{if } V_1 = V_1', t^o \\ V_1 & \text{if } t \notin D(V_1) \end{cases}$$

Ordering on kinds:

$$\begin{array}{rcl}
V_1 \leq V_2 & \text{iff} & \forall t, Var(t, V_2) \leq Var(t, V_1) \\
(M_1; V_1) \leq (M_2; V_2) & \text{iff} & M_1 \subseteq M_2 \text{ and } V_1 \leq V_2 \\
T^a \to \nu_1 \leq T^b \to \nu_2 & \text{iff} & b \leq a \text{ and } \nu_1 \leq \nu_2
\end{array}$$

C Typing rules

C.1 Context Rules

$(start)$

$$\frac{}{\epsilon \vdash *}$$

$(type\ var)$

$$\frac{\begin{array}{c} , \vdash * \\ t \notin dom(,) \end{array}}{,, t : \{t^+\} \vdash *}$$

$$(\mathit{row\ var}) \qquad \frac{\begin{array}{c}, \vdash R_1 :: S_1 \to (M_1;\, V_1) \\ S_0 \to (M_0;\, V_0) \leq S_1 \to (M_1;\, V_1) \\ D(V_0) \subseteq \mathit{dom}(,\ ) \\ r \notin \mathit{dom}(,\ )\end{array}}{,\ , (r <:_w R_1 :: S_0 \to (M_0;\, V_0)) \vdash *}$$

$$(\mathit{exp\ var}) \qquad \frac{\begin{array}{c}, \vdash \tau : V \\ x \notin \mathit{dom}(,\ )\end{array}}{,\ , x : \tau \vdash *}$$

$$(\mathit{weakening}) \qquad \frac{\begin{array}{c}, _1, , _2 \vdash A \\ , _1, a, , _2 \vdash *\end{array}}{, _1, a, , _2 \vdash A}$$

$$\text{where } a ::= x : \tau \mid t : V \mid r <:_w R :: \kappa$$

C.2 Rules for type expressions

$$(\mathit{type\ proj}) \qquad \frac{\begin{array}{c}, \vdash * \\ t : \{t^+\} \in ,\end{array}}{, \vdash t : \{t^+\}}$$

$$(\mathit{type\ arrow}) \qquad \frac{\begin{array}{c}, \vdash \tau_1 : V_1 \\ , \vdash \tau_2 : V_2\end{array}}{, \vdash \tau_1 \to \tau_2 : \mathit{Merge}(\overline{V_1}, V_2)}$$

$$(\mathit{pro}) \qquad \frac{, , t : \{t^+\} \vdash R :: (M;\, V)}{, \vdash \mathbf{pro}\, t\mathbf{.}R : \mathit{Invar}(V \setminus t)}$$

$$(\mathit{cov\ object}) \qquad \frac{\begin{array}{c}, , t : \{t^+\} \vdash R :: (M;\, V) \\ \mathit{Var}(t, V) \in \{+, ?\}\end{array}}{, \vdash \mathbf{obj}\, t\mathbf{.}R : V \setminus t}$$

$$(\mathit{non\ cov\ object}) \qquad \frac{\begin{array}{c}, , t : \{t^+\} \vdash R :: (M;\, V) \\ \mathit{Var}(t, V) \in \{o, -\}\end{array}}{, \vdash \mathbf{obj}\, t\mathbf{.}R : \mathit{Invar}(V \setminus t)}$$

$$(\mathit{exist}) \qquad \frac{\begin{array}{c}, , r <:_w R :: \kappa \vdash \tau : V_1 \\ \kappa = S \to (M;\, V_2)\end{array}}{, \vdash \exists (r <:_w R :: \kappa)\tau : \mathit{Merge}(V_1, V_2)}$$

C.3 Rules for rows

(*row proj*)

$$\frac{\begin{array}{c}, \vdash * \\ r <:_w R :: \kappa \in dom(,\ )\end{array}}{, \vdash r :: \kappa}$$

(*empty row*)

$$\frac{, \vdash *}{, \vdash \langle\!\langle\rangle\!\rangle :: (M; \emptyset)}$$

(*row label*)

$$\frac{\begin{array}{c}, \vdash R :: S^i \to (M; V) \\ N \subseteq M \qquad i \in \{0,1\}\end{array}}{, \vdash R :: S^i \to (N; V)}$$

(*row fn abs*)

$$\frac{, , t : \{t^+\} \vdash R :: (M; V)}{, \vdash \lambda t.\, R :: T^{Var(t,V)} \to (M; V \backslash t)}$$

(*row fn app cov*)

$$\frac{\begin{array}{c}, \vdash R :: T^+ \to (M; V_1) \\ , \vdash \tau : V_2\end{array}}{, \vdash R\tau :: (M; Merge(V_1, V_2))}$$

(*row fn app contra*)

$$\frac{\begin{array}{c}, \vdash R :: T^- \to (M; V_1) \\ , \vdash \tau : V_2\end{array}}{, \vdash R\tau :: (M; Merge(V_1, \overline{V_2}))}$$

(*row fn app inv*)

$$\frac{\begin{array}{c}, \vdash R :: T^o \to (M; V_1) \\ , \vdash \tau : V_2\end{array}}{, \vdash R\tau :: (M; Merge(V_1, Invar(V_2)))}$$

(*row fn app vac*)

$$\frac{\begin{array}{c}, \vdash R :: T^? \to (M; V_1) \\ , \vdash \tau : V_2\end{array}}{, \vdash R\tau :: (M; V_1)}$$

(*row ext*)

$$\frac{\begin{array}{c}, \vdash R :: (\{\vec{m}, m\}, V_1) \\ , \vdash \tau : V_2\end{array}}{, \vdash \langle\!\langle R \,|\, m : \tau \rangle\!\rangle :: (\{\vec{m}\}; Merge(V_1, V_2))}$$

C.4 Subtyping rules for types

$(<:\ \mathit{type\ refl})$
$$\frac{,\ \vdash \tau : V}{,\ \vdash \tau <: \tau}$$

$(<: \rightarrow)$
$$\frac{\begin{array}{c},\ \vdash \tau_1' <: \tau_1 \\ ,\ \vdash \tau_2 <: \tau_2'\end{array}}{,\ \vdash \tau_1 \rightarrow \tau_2 <: \tau_1' \rightarrow \tau_2'}$$

$(<: \mathit{obj})$
$$\frac{\begin{array}{c},\ ,\ t : \{t^+\} \vdash R_1 <:_B R_2 \\ ,\ ,\ t : \{t^+\} \vdash R_2 :: (M; V) \\ Var(t, V) \in \{?, +\}\end{array}}{,\ \vdash \mathbf{probj}\, t\mathbf{.} R_1 <: \mathbf{obj}\, t\mathbf{.} R_2}$$

$(<: \mathit{trans})$
$$\frac{\begin{array}{c},\ \vdash \tau_1 <: \tau_2 \\ ,\ \vdash \tau_2 <: \tau_3\end{array}}{,\ \vdash \tau_1 <: \tau_3}$$

C.5 Subtyping Rules for rows

$(<:\ \mathit{row\ refl})$
$$\frac{,\ \vdash R :: \kappa}{,\ \vdash R <:_B R}$$

$(\mathit{row\ proj\ bound})$
$$\frac{\begin{array}{c},\ \vdash * \\ r <:_w R :: \kappa \in\ ,\end{array}}{,\ \vdash r <:_w R}$$

$(<: \lambda)$
$$\frac{\begin{array}{c},\ ,\ t : \{t^+\} \vdash R_1 <:_B R_2 \\ ,\ ,\ t : \{t^+\} \vdash R_2 :: \nu\end{array}}{,\ \vdash \lambda t.\, R_1 <:_B \lambda t.\, R_2}$$

$(<: \mathit{app\ cong})$
$$\frac{\begin{array}{c},\ \vdash R_1 <:_B R_2 \\ ,\ \vdash R_2 :: T^a \rightarrow \nu \\ ,\ \vdash \tau : V\end{array}}{,\ \vdash R_1 \tau <:_B R_2 \tau}$$

$(<: \mathit{app\ cov})$
$$\frac{\begin{array}{c},\ \vdash R_1 <:_B R_2 \\ ,\ \vdash R_2 :: T^+ \rightarrow \nu \\ ,\ \vdash \tau_1 <: \tau_2\end{array}}{,\ \vdash R_1 \tau_1 <:_{B+d} R_2 \tau_2}$$

$(<:\ app\ contra)$
$$\frac{\begin{array}{c}, \vdash R_1 <:_B R_2 \\ , \vdash R_2 :: T^- \to \nu \\ , \vdash \tau_2 <: \tau_1\end{array}}{, \vdash R_1\tau_1 <:_{B+d} R_2\tau_2}$$

$(<:\ app\ vac)$
$$\frac{\begin{array}{c}, \vdash R_1 <:_B R_2 \\ , \vdash R_2 :: T^? \to \nu \\ , \vdash \tau_1 : V_1 \qquad , \vdash \tau_2 : V_2\end{array}}{, \vdash R_1\tau_1 <:_B R_2\tau_2}$$

$(<:\ cong)$
$$\frac{\begin{array}{c}, \vdash R_1 <:_B R_2 \qquad , \vdash \tau : V \\ , \vdash \langle\!\langle R_i \,|\, m : \tau \rangle\!\rangle :: \nu_i \qquad i \in \{1, 2\}\end{array}}{, \vdash \langle\!\langle R_1 \,|\, m : \tau \rangle\!\rangle <:_B \langle\!\langle R_2 \,|\, m : \tau \rangle\!\rangle}$$

$(<:\ d)$
$$\frac{\begin{array}{c}, \vdash R_1 <:_B R_2 \qquad , \vdash \tau_1 <: \tau_2 \\ , \vdash \langle\!\langle R_i \,|\, m : \tau_i \rangle\!\rangle :: \nu_i \qquad i \in \{1, 2\}\end{array}}{, \vdash \langle\!\langle R_1 \,|\, m : \tau_1 \rangle\!\rangle <:_{B+d} \langle\!\langle R_2 \,|\, m : \tau_2 \rangle\!\rangle}$$

$(<:\ w)$
$$\frac{\begin{array}{c}, \vdash R_1 <:_B R_2 \\ , \vdash \langle\!\langle R_1 \,|\, m : \tau \rangle\!\rangle :: \nu\end{array}}{, \vdash \langle\!\langle R_1 \,|\, m : \tau \rangle\!\rangle <:_{B+w} R_2}$$

$(<:\ trans)$
$$\frac{\begin{array}{c}, \vdash R_1 <:_B R_2 \\ , \vdash R_2 <:_{B'} R_3\end{array}}{, \vdash R_1 <:_{B+B'} R_3}$$

Type and Row Equality Type or row expressions that differ only in names of bound variables are considered identical. Additional equations between types and rows arise as a result of β-reduction, written $\to_\beta$, or β-conversion, written $\leftrightarrow_\beta$.

$(row\ \beta)$
$$\frac{, \vdash R :: \kappa, \quad R \to_\beta R'}{, \vdash R' :: \kappa}$$

$(type\ \beta)$
$$\frac{, \vdash \tau : V, \quad \tau \to_\beta \tau'}{, \vdash \tau' : V}$$

$(type\ eq)$
$$\frac{, \vdash e : \tau, \quad \tau \leftrightarrow_\beta \tau', \quad , \vdash \tau' : V}{, \vdash e : \tau'}$$

$(<:\ \beta\ right)$
$$\frac{, \vdash R_1 <:_B (\lambda t.\, R_2)\tau_2}{, \vdash R_1 <:_B [\tau_2/t]R_2}$$

$(<:\ \beta\ left)$
$$\frac{, \vdash (\lambda t.\, R_1)\tau_1 <:_B R_2}{, \vdash [\tau/t]R_1 <:_B R_2}$$

C.6 Rules for assigning types to terms

$(exp\ proj)$
$$\frac{\begin{array}{c}, \vdash * \\ x : \tau \in ,\end{array}}{, \vdash x : \tau}$$

$(subsumption)$
$$\frac{, \vdash e : \tau_1, \quad , \vdash \tau_1 <: \tau_2}{, \vdash e : \tau_2}$$

$(exp\ abs)$
$$\frac{, , x : \tau_1 \vdash e : \tau_2}{, \vdash \lambda x. e : \tau_1 \rightarrow \tau_2}$$

$(exp\ app)$
$$\frac{, \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad , \vdash e_2 : \tau_1}{, \vdash e_1\, e_2 : \tau_2}$$

$(empty\ pro)$
$$\frac{, \vdash *}{, \vdash \langle\rangle : \mathbf{pro}\, t\mathbf{.}\langle\!\langle\rangle\!\rangle}$$

$(pro\ ext)$
$$\frac{\begin{array}{l}, \vdash e_1 : \mathbf{pro}\, t\mathbf{.}R \\ , , t : \{t^+\} \vdash R :: (\{m\}; V) \\ , , I_r \vdash e_2 : [\mathbf{pro}\, t\mathbf{.}rt/t](t \rightarrow \tau) \qquad r \notin V(\tau)\end{array}}{, \vdash \langle e_1 \longleftrightarrow m{=}e_2\rangle : \mathbf{pro}\, t\mathbf{.}\langle\!\langle R \,|\, m : \tau\rangle\!\rangle}$$
where $I_r = r <:_w \lambda t.\langle\!\langle R \,|\, m : \tau\rangle\!\rangle :: T^0 \rightarrow (\emptyset;\, Invar(V_\Gamma))$.

$(pro\ over)$
$$\frac{\begin{array}{l}, \vdash e_1 : \mathbf{pro}\, t\mathbf{.}R \\ , , t : \{t^+\} \vdash R <:_w \langle\!\langle m : \tau\rangle\!\rangle \\ , , I_r \vdash e_2 : [\mathbf{pro}\, t\mathbf{.}rt/t](t \rightarrow \tau)\end{array}}{, \vdash \langle e_1 \leftarrow m{=}e_2\rangle : \mathbf{pro}\, t\mathbf{.}R}$$
where $I_r = r <:_w \lambda t.\, R :: T^0 \rightarrow (\emptyset;\, Invar(V_\Gamma))$.

$(probj \Leftarrow)$
$$\frac{\begin{array}{c}, \vdash e : \mathbf{probj}\, t\mathbf{.}R \\ , , t : \{t^+\} \vdash R <:_w \langle\!\langle m : \tau\rangle\!\rangle\end{array}}{, \vdash e \Leftarrow m : [\mathbf{probj}\, t\mathbf{.}R/t]\tau}$$

$(\exists <:\ intro)$
$$\frac{\begin{array}{c}, \vdash R_1 :: \kappa \\ , \vdash R_1 <:_w R \\ , \vdash e : [R_1/r]\tau\end{array}}{, \vdash \{\!|r <:_w R :: \kappa = R_1,\ e|\!\} : \exists(r <:_w R :: \kappa)\tau}$$

$(\exists <:\ elim)$
$$\frac{\begin{array}{c}, \vdash e_1 : \exists(r <:_w R :: \kappa)\tau \\ , , r <:_w R :: \kappa,\, x : \tau \vdash e_2 : \sigma \\ , \vdash \sigma : V\end{array}}{, \vdash Abstype\ r <:_w R :: \kappa\ with\ x : \tau\ is\ e_1\ in\ e_2 : \sigma}$$

D Overview of soundness proof

We prove the soundness of our type system with respect to the operational semantics given in Section 3.2. We first prove that evaluation preserves type; this property is traditionally called subject reduction. We then show that no typable expression evaluates to *error* using the evaluation rules given in [FHM94], extended in a straightforward manner to account for the new encapsulation primitives. This fact guarantees that we have no message-not-understood errors for expressions with **pro** types.

The proof begins with two lemmas about substitution for row and type variables. We then prove a normal form lemma that allows us to restrict our attention to derivations of a certain form, simplifying later proofs. Lemmas D.5 and D.6 give us subject reduction for the bookkeeping rules and β-reduction, respectively. Lemmas D.7, D.8, and D.9 imply subject reduction for $(\Leftarrow)$-reduction, while Lemmas D.1 and D.6 give us subject reduction for $(Class)$-reduction.

In the remainder of this section, we use meta-variable U to refer to either a row or type expression, meta-variable γ to refer to arbitrary kinds, and meta-judgement $, \vdash U : -\gamma$ to indicate either "row has kind γ" or "type is well-formed."

The first two lemmas are substitution lemmas that are used to specialize **pro** types to contain additional methods and to prove $(Class)$ reduction sound.

Lemma D.1 (Row Substitution)

If $\quad , , r <:_w R' :: \kappa, ,' \vdash A, \quad , \vdash R <:_B R', \quad$ *and* $\quad , \vdash R :: \kappa,$

then $\quad , , [R/r], ,' \vdash A',$

where

- *if* $A \equiv *$, *then* $A' \equiv *$
- *if* $A \equiv U : -\gamma$, *then* $A' \equiv [R/r]U : -\gamma$
- *if* $A \equiv U_1 <:_B U_2$, *then* $A' \equiv [R/r]U_1 <:_B [R/r]U_2$
- *if* $A \equiv e : \sigma$, *then* $A' \equiv [R/r]e : [R/r]\sigma$

Lemma D.2 (Type Substitution) *If* $, , t : T, ,' \vdash A$ *and* $, \vdash \tau : V$ *then*

- *if* $A \equiv *$, *then* $, , [\tau : V/t], ,' \vdash *$
- *if* $A \equiv U : -\gamma$, *then* $, , [\tau : V/t], ,' \vdash [\tau/t]U : -\gamma$
- *if* $A \equiv U_1 <:_B U_2$, *then* $, , [\tau : V/t], ,' \vdash [\tau/t]U_1 <:_B [\tau/t]U_2$

The type and row equality rules introduce many non-essential judgement derivations, which unnecessarily complicate derivation analysis. We therefore restrict our attention to $\vdash_N$-derivations, which we define as those derivations in which the only appearance of a type or row equality rule is as $(\beta <: right)$ immediately following an occurrence of a subtyping application rule $(<:\ app\ *)$ where the left-hand row function expression is a row variable, or as $(type\ eq)$ immediately before an occurrence of $(\exists <: intro)$. The τnf of a type or row expression is its normal form with respect to β-reduction. The τnf of a term expression e is just e. Since we are only interested in types and rows in τnf, the following lemma shows we can find a $\vdash_N$-derivation for any judgement of interest.

Lemma D.3 (Normal Form for Derivations) *If* $, \vdash A$ *is derivable, then so is* $\tau nf(,) \vdash_N \tau nf(A)$.

The proof of this lemma is by induction on the derivation of $, \vdash A$. Occurrences of equality rules may be eliminated in the $\vdash_N$-derivation because two row or type expressions related via β-reduction must have the same τnf. The cases for the row application rules $(row\ fn\ app\ *)$ follows from the somewhat surprising fact that if we may derive that a normal-form row function $\lambda t. R'$ applied to a normal-form type τ is well-formed, then we may show that $[\tau/t]R'$ is well-formed without using the $(row\ \beta)$ typing rule. The cases for $(<:\ app\ *)$ are similar.

From this point on, we will only concern ourselves with contexts and type or row expressions that are in τnf. This limitation is not severe, since any term that has a type has a type in τnf. Future analyses of derivations will consider only $\vdash_N$-derivations, since its restriction on equality rules greatly simplifies the proofs.

The following lemma is used to show that the (*switch ext ov*) reduction rule preserves types.

Lemma D.4 *If* $, , p <:_w \lambda t.\, R :: T \rightarrow (M_p; V), ,' \vdash A$ *and* $, , r <:_w \lambda t.\langle\!\langle R \,|\, m : \tau \rangle\!\rangle :: T \rightarrow (M_r; V) \vdash A$ *are both derivable,* $r \notin dom(, ')$ *and* $M_p \subseteq M_r$*, then* $, , r <:_w \lambda t.\langle\!\langle R \,|\, m : \tau \rangle\!\rangle :: T \rightarrow (M_r; V), [r/p], ' \vdash [r/p]A$ *is also derivable, where* A *is any judgement not involving strictly* σ*-types.*

The proof of this lemma is very sensitive to the form of the (*pro ext*) proof rule.

The next four lemmas show that the various components of the $\xrightarrow{eval}$ relation preserve expression types. Lemma D.5 is the first of these, showing that the $(\xrightarrow{book})$ relation has the necessary property.

Lemma D.5 *If* $, \vdash e : \tau$ *is derivable, and* $e \xrightarrow{book} e'$*, then* $, \vdash e' : \tau$ *is derivable.*

The proof of Lemma D.5 consists of two parts: the first shows that a derivation from $, \vdash e : \tau$ can only depend on the form of τ, not on the form of e. More formally, if $, \vdash C[e] : \tau$ is derived from $, ' \vdash e : \sigma$ and $, ' \vdash e' : \sigma$ is also derivable, then so is $, \vdash C[e'] : \tau$. This fact is easily seen by an inspection of the typing rules. The second part shows that if $, \vdash e : \tau$ is derivable, and $e \xrightarrow{book} e'$ by e matching the left-hand side of one of the $(\xrightarrow{book})$ axioms, then $, \vdash e' : \tau$ is also derivable. This fact follows from a case analysis of the four $(\xrightarrow{book})$ axioms. Lemma D.4 is essential for the (*switch ext ov*) case.

The fact that (β)-reduction preserves expression types is an immediate consequence of the following lemma:

Lemma D.6 (Expression Substitution) *If* $, , x : \tau_1, ,' \vdash e_2 : \tau_2$ *and* $, \vdash e_1 : \tau_1$ *are both derivable, then so is* $, , ,' \vdash [e_1/x]e_2 : \tau_2$.

Lemma D.6 is proved by induction on the derivation of $, , x : \tau_1, ,' \vdash e_2 : \tau_2$.

The next three lemmas together imply that $(\Leftarrow)$-reduction preserves type. The first is the key lemma in showing subject reduction for messages sent to expressions with **pro** type. The second, which guarantees that the variance annotations properly track variance, is essentially for showing the soundness of sealing **obj** types to **pro** types. The third implies subject reduction for expressions with **obj** type.

Lemma D.7 (Method Bodies are Type Correct) *If* $, \vdash_N \langle e_1 \hookleftarrow m = e_2 \rangle : \mathbf{pro}\, t.R$ *is derivable in such a way that the last rule in the derivation is not* (*type eq*)*, then there exists a unique type* τ *such that* $, , t : \{t^+\} \vdash_N R <:_w \langle\!\langle m : \tau \rangle\!\rangle$ *and* $, \vdash_N e_2 : [\mathbf{pro}\, t.R/t](t \rightarrow \tau)$ *are both derivable. Furthermore, if* $\hookleftarrow = \leftarrow$ *then* $, \vdash_N e_1 : \mathbf{pro}\, t.R$ *is also derivable.*

Lemma D.8 *If* $, , t : \{t^+\}, ,_\tau \vdash_N U_1 <:_B U_2$, $, \vdash_N U_2 : -\gamma_2$, $, \vdash_N \tau_1 <: \tau_2$, *and* $, \vdash_N \tau_i : V_i$ *for* $i \in \{1, 2\}$ *are all derivable, then*

- *if* $Var(t, \gamma) = ?$ *then*
  $, , ,_\tau \vdash_N [\tau_i/t]U_1 <:_B [\tau_j/t]U_2$
- *if* $Var(t, \gamma) = +$ *then*
  $, , ,_\tau \vdash_N [\tau_1/t]U_1 <:_{w,d} [\tau_2/t]U_2$
- *if* $Var(t, \gamma) = -$ *then*
  $, , ,_\tau \vdash_N [\tau_2/t]U_1 <:_{w,d} [\tau_1/t]U_2$

where $,_\tau$ *is a context listing only type variables.*

Lemma D.9 (Object Types Come From Pro Types) *If* $, \vdash_N \langle e_1 \leftarrow\!\!+ m = e_2 \rangle : \mathbf{obj}\,t\,\boldsymbol{.}R$ *is derivable, then there exists a type* $\mathbf{pro}\,t\,\boldsymbol{.}R'$ *such that*

$$
\begin{array}{l}
, \vdash_N \langle e_1 \leftarrow\!\!+ m = e_2 \rangle : \mathbf{pro}\,t\,\boldsymbol{.}R' \\
, \vdash_N \mathbf{pro}\,t\,\boldsymbol{.}R' <: \mathbf{obj}\,t\,\boldsymbol{.}R
\end{array}
$$

are both derivable.

Theorem D.10 (Subject Reduction) *If* $, \vdash e : \tau$ *is derivable, and* $e \xrightarrow{eval} e'$, *then* $, \vdash e' : \tau$ *is also derivable.*

The proof is similar in outline to that of Lemma D.5; it reduces to showing that each of the basic evaluation steps preserves the type of the expression being reduced. The $(\xrightarrow{book})$ case follows from Lemma D.5, the (β) case from Lemma D.6, the $(\Leftarrow)$ case from Lemmas D.7, D.8, and D.9, and the $(Class)$ case from Lemmas D.1 and D.6.

Theorem D.11 (Type Soundness) *If the judgement* $\epsilon \vdash e : \tau$ *is derivable, then* $eval(e) \neq error$, *where the function* $eval$ *is as in [FHM94], extended with rules for* $(Class)$ *reduction.*