Classes = Objects + Data Abstraction

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

October 24, 1995

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++'s `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 (a 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/or 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 [AC94, 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 first explain the basic principles by extending a core object calculus (developed for this purpose but previously described in [FM95]) with a standard higher-order data abstraction mechanism as in [MP88, CW85]. Then we devise a special-purpose syntax

*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.

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. Then 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 behavioural similariy 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. On the other hand, there are a number of details involving subtype assertions about abstract types, extensions to abstract types, covariance and contravariance of methods, and so on that make the exact details of the system relatively subtle. The main idea is that we may regard a class, which consists of a type (of objects) together with an implementation as an abstract data type. This is plausible from the outset since an abstract data type is well-known to consist of a type together with implementations for operations. However, in order to allow inheritance (the extension of abstract types with additional operations) and subtyping, we must begin with a relatively flexible underlying object calculus and a suitably general abstraction mechanism. More specifically, the type system must allow us to extend objects with new operations (or replace existing methods), provide subtyping on interface types, and support abstract type declarations that include subtype specifications, negative information (absence of methods), and variance information. The abstract data type declarations must include such information so that they are extensible in essentially the same way as their underlying representations. Subject to these relatively general constraints, the core object calculus we use in this paper could be replaced with other styles of object calculi, such as developed in [**?**].

The main contribution of this paper is to show how to obtain a family of variations and extensions of C++-like and Eiffel-like class mechanisms by combining a core object system with an accepted form of data abstraction mechanism. In addition, we identify a set of object operations that are needed to justify this view of classes and prove the type soundness of the resulting formal calculus of objects and abstraction. While some of the improvements in our core calculus over previous papers ([FHM94, FM95]) may have independent interest, they are motivated primarily by the technical demands of our analysis of class constructs. An important detail for readers of [FHM94] is that the "class" types of that paper were interface types, not classes in the sense of type of objects with implementation. 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 implemention of the class,

Protected methods are only accessible in the implemention 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.

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 mneumonics, we follow a systematic naming convention where, for class $\mathtt{A}$, 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 each class is recursively defined; the type is a fixed-point of a function from types to types and point-wise subtyping between type functions is the critical relation between interfaces.

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

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

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.

The analogous interfaces for `ColorPoint` are written using a free row (function from types to finite sets of method/type pairs) variable `p`, which will be bound to the abstract type-function for points in the scope where the `ColorPoint` interfaces will appear:

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

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` (see below) will guarantee that methods $\mathtt{c} : \mathtt{Color}$, $\mathtt{setC} : \mathtt{Color} \rightarrow \mathtt{t}$, $\mathtt{getC} : \mathtt{Color}$ 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, of subtype-bounded existential type, of the form

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

consisting of the private interface $\mathtt{P}_{\mathtt{priv}}$ of points and a constructor function $\mathtt{ConsImp}_{\mathtt{p}}$ that might use an initial value for the `x`-coordinate, for example, to return a 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 elipsis $(\ldots)$ 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"

$$
\begin{array}{rl}
\texttt{Abstype} & \texttt{p} <: \texttt{P}_{\texttt{prot}} :: \texttt{T} \rightarrow \{\texttt{c}, \texttt{getC}, \texttt{setC}\} \\
\texttt{with} & \texttt{newPoint} : \texttt{Int} \rightarrow \mathbf{pro}\, \texttt{u}.\texttt{p}\, \texttt{u} \\
\texttt{is} & \{\!|\texttt{p} <: \texttt{P}_{\texttt{prot}} :: (\texttt{T} \rightarrow \{\texttt{c}, \texttt{getC}, \texttt{setC}\}) = \texttt{P}_{\texttt{priv}},\ \ \texttt{Imp}_{\texttt{p}}|\!\} \\
\texttt{in} &
\end{array}
$$

$$
\begin{array}{rl}
\texttt{Abstype} & \texttt{cp} <: \texttt{CP}_{\texttt{prot}} :: \texttt{T} \rightarrow \emptyset \\
\texttt{with} & \texttt{newColorPoint} : \texttt{Int} \rightarrow \texttt{Color} \rightarrow \mathbf{pro}\, \texttt{u}.\texttt{cp}\, \texttt{u} \\
\texttt{is} & \{\!|\texttt{cp} <: \texttt{CP}_{\texttt{prot}} :: (\texttt{T} \rightarrow \emptyset) = \texttt{CP}_{\texttt{priv}},\ \ \texttt{Imp}_{\texttt{cp}}|\!\} \\
\texttt{in} &
\end{array}
$$

$$
\begin{array}{rl}
\texttt{Abstype} & \texttt{p} <: \texttt{P}_{\texttt{pub}} :: \texttt{T} \rightarrow \emptyset \\
\texttt{with} & \texttt{newPoint} : \texttt{Int} \rightarrow \mathbf{obj}\, \texttt{t}.\texttt{p}\, \texttt{t} \\
\texttt{is} & \{\!|\texttt{p} <: \texttt{P}_{\texttt{pub}} :: (\texttt{T} \rightarrow \emptyset) = \texttt{p},\ \ \texttt{newPoint}|\!\} \\
\texttt{in} &
\end{array}
$$

$$
\begin{array}{rl}
\texttt{Abstype} & \texttt{cp} <: \texttt{CP}_{\texttt{pub}} :: \texttt{T} \rightarrow \emptyset \\
\texttt{with} & \texttt{newColorPoint} : \texttt{Int} \rightarrow \texttt{Color} \rightarrow \mathbf{obj}\, \texttt{t}.\texttt{cp}\, \texttt{t} \\
\texttt{is} & \{\!|\texttt{cp} <: \texttt{P}_{\texttt{pub}} :: (\texttt{T} \rightarrow \emptyset) = \texttt{cp},\ \ \texttt{newColorPoint}|\!\} \\
\texttt{in} & \\
 & \langle\texttt{Program}\rangle
\end{array}
$$

Figure 1: Nest 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 implementation of `ColorPoint` similarly has the form

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

where the constructor $\texttt{ConsImp}_{\texttt{cp}}$, 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 a 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. However, the essential hiding mechanism and export of interfaces for derived classes and clients remain essentially the same.

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 given by the grammer

$$\mathtt{e} ::= \mathtt{x} \mid \mathtt{c} \mid \lambda\mathtt{x}.\,\mathtt{e} \mid \mathtt{e}_1\mathtt{e}_2 \mid \langle\rangle \mid \mathtt{e} \Leftarrow \mathtt{m} \mid \langle \mathtt{e}_1 \longleftarrow\!\!\!+ \mathtt{m}{=}\mathtt{e}_2 \rangle \mid \langle \mathtt{e}_1 \leftarrow \mathtt{m}{=}\mathtt{e}_2 \rangle$$

where $\mathtt{x}$ may be any variable, $\mathtt{c}$ is a constant symbol (such as a "built-in" function), $\lambda\mathtt{x}.\,\mathtt{e}$ is a lambda abstraction (function expression) and $\mathtt{e}_1\,\mathtt{e}_2$ is function application. The object forms are:

$\langle\rangle$	the empty prototype or object
$\mathtt{e} \Leftarrow \mathtt{m}$	send message $\mathtt{m}$ to prototype or object $\mathtt{e}$
$\langle \mathtt{e}_1 \longleftarrow\!\!\!+ \mathtt{m}{=}\mathtt{e}_2 \rangle$	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 and we extend the core calculus with two encapsulation primitives:

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

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 $\leftarrow\!\!\circ$ may be either $\longleftarrow\!\!+$ or $\longleftarrow$. We also need various bookkeeping rules to access methods defined within $\mathtt{e_1}$. These bookkeeping rules appear in Appendix A; they are explained in full in [FHM94]. We also need reduction rule (*Abstype*) for evaluating our abstractions:

$$\mathtt{Abstype\ r} <:_{\mathtt{w}} \mathtt{R}\ ::\kappa\ \mathtt{with\ x}:\tau\ \mathtt{is}\ \{\!|\mathtt{r} <:_{\mathtt{w}} \mathtt{R}\ ::\kappa = \mathtt{R}',\ \mathtt{e_1}|\!\}\ \mathtt{in\ e_2} \xrightarrow{eval} [\mathtt{R}'/\mathtt{r}, \mathtt{e_1}/\mathtt{x}]\mathtt{e_2}$$

(* Is this better, or is the form with I_r better? *) (* Better I think, since the "macro" status of I_r always confused me – jcm *)

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. Row expressions appear as subexpressions of type expressions, with rows and types distinguished by kinds. Intuitively, the elements of kind $(\{\vec{m}\}; V)$ are lists of method names and types that do *not* include the method names in $\{\vec{m}\}$. Futhermore, the free type variables appearing in the elements of this kind have the variances indicated by V. We must statically know that some methods do not appear in a row in order to guarantee that methods are not multiply defined. The variance information is necessary to determine subtyping judgements between types of the form $\mathbf{pro}\,t.R$ and $\mathbf{obj}\,t.R'$, where either or both of R and R' may contain row variables. To avoid unnecessary repetition in our presentation, we use the meta-variable *prob* for either **obj** or **pro**. Row expressions with kind $T \rightarrow (\{\vec{m}\}; V)$ are used to infer a form of higher-order polymorphism for method bodies and to provide type interfaces to encapsulated implementations.

The elements of type $\mathbf{prob}\,t.\langle\!\langle m_1 : \tau_1, \ldots, m_k : \tau_k\rangle\!\rangle$ are objects e such that for $1 \le i \le k$, the result of $e \Leftarrow m_i$ is a value of type τ_i Since the bound type variable t may appear free within types $\tau_1, \ldots, \tau_k$, when we say that $e \Leftarrow m_i$ has type τ_i, we mean type τ_i with any free occurrences of t in τ_i replaced by the type $\mathbf{prob}\,t.\langle\!\langle m_1 : \tau_1, \ldots, m_k : \tau_k\rangle\!\rangle$. Thus $\mathbf{prob}\,t.\langle\!\langle\ldots\rangle\!\rangle$ is a special form of recursively-defined type.

The typing rule for message send is

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

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 $\Gamma,\, t:\{t^+\} \vdash R <:_w \langle\!\langle m:\tau\rangle\!\rangle$ 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:

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

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

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 $\langle\!\langle R \,|\, m : \tau \rangle\!\rangle$, which permits us to override methods in objects with partially abstract types.

Since sealing a prototype e requires no change to e, we would have liked to be able to use the following rule in conjunction with a subsumption rule to "seal" prototypes:

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

where the judgement $\Gamma \vdash \mathbf{obj}\, u\,\mathbf{.}R : T$ denotes that type $\mathbf{obj}\, u\,\mathbf{.}R$ is well formed. Unfortunately, this rule is problematic when the variable t appears contravariantly in R. Hence we need a more complicated rule that has the same form as the rule for deriving subtyping judgements on **obj** types. Because of this similarity, we may combine the sealing rule and the **obj** subtyping rule to produce:

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

The meta-variable **prob** is **pro** when this rule is used to seal prototypes. When used simply for subtyping between object types, **prob** is **obj**. Because we need to infer subtyping judgements between object types whose rows contain row variables, we must have a judgement that supplies the necessary variance information. In particular, we need to infer that the bound type t appears covariantly in the row from the right hand side of the subtyping judgement. The function $Var(t, V)$ looks up the type variable t in the variance set V inferred for row R_2. If $Var(t, V)$ is $?$, then t does not appear in R_2, and hence is vacuously covariant. If $Var(t, V)$ is $+$, then t appears covariantly. In either case, the subtyping judgment is valid.

Notice that since the subscript B on the subtype judgment is unconstrained, this rule gives us subtyping in both width and depth for **obj** types

This rule also has the somewhat unfortunate consequence that we cannot seal prototypes to **obj** types containing 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:

$$(\exists <:\ intro)\quad \frac{\Gamma \vdash R_1 :: S_1 \to (M_1;\ V_1) \qquad \Gamma \vdash R_1 <:_w R \qquad \Gamma \vdash e : [R_1/r]\tau}{\Gamma \vdash \{\!|\, r <:_w R :: \kappa = R_1,\ e \,|\!\} : \exists (r <:_w R :: \kappa)\tau}$$

$$\text{where } \kappa = S_1 \to (M_1;\, V_1)\,.$$

(* Explain this rule *) The rule for using existential types is standard (as well).
(** The full type system appears in Appendix C. Summary of other rules??? **)

3.4 Examples

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

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

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`:

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

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 in one family of object-oriented languages, 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 that deserves further consideration.

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. Specifically, the clauses giving possible subtypes of `p` are type-checked using the assumptions $\mathtt{p} <: \mathtt{P\_prot} :: \mathtt{kind\_p}$ and $\mathtt{newP} : \mathtt{pars\_p} \to \mathbf{pro}\, \mathtt{u.p\,u}$, while the program is type-checked using the assumptions $\mathtt{p} <: \mathtt{P\_pub} :: \mathtt{T} \to \emptyset$ and $\mathtt{newP} : \mathtt{pars\_p} \to \mathbf{obj}\, \mathtt{u.p\,u}$. This makes it possible for the implementation of derived classes (the classes occurring before `in Program end`) 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. Since there is no restriction on Q_pub, for example, this could be declared so that $\mathbf{obj}\, u.q\, u <: \mathbf{obj}\, u.p\, u$, or not. As a result, we have the analogs of both C++ "public base classes" in which inheritance results in a subtype and and "private base classes" where it does not.

4 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 4.5 and 4.6 give us subject reduction for the bookkeeping rules and β-reduction, respectively. Lemmas 4.7, 4.8, and 4.9 imply subject reduction for ($\Leftarrow$)-reduction, while Lemmas 4.1 and 4.6 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 $\Gamma \vdash U : -\gamma$ 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 4.1 (Row Substitution)

$$\begin{array}{ll} \textit{If} & \Gamma,\, r <:_w R' :: \kappa,\, \Gamma' \vdash A, \quad \Gamma \vdash R <:_B R', \quad \textit{and} \quad \Gamma \vdash R :: \kappa, \\ \textit{then} & \Gamma,\, [R/r]\Gamma' \vdash A', \\ \textit{where} & \end{array}$$

- *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 4.2 (Type Substitution) *If* $\Gamma, t : T, \Gamma' \vdash A$ *and* $\Gamma \vdash \tau : V$ *then*

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

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

The proof of this lemma is by induction on the derivation of $\Gamma \vdash A$. 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* β) 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 4.4 *If* $\Gamma, p <:_w \lambda t.\, R :: T \to (M_p;\, V), \Gamma' \vdash A$ *and* $\Gamma, r <:_w \lambda t.\langle\!\langle R \mid m : \tau \rangle\!\rangle :: T \to (M_r;\, V) \vdash A$ *are both derivable,* $r \notin dom(\Gamma')$ *and* $M_p \subseteq M_r$, *then* $\Gamma, r <:_w \lambda t.\langle\!\langle R \mid m : \tau \rangle\!\rangle :: T \to M_r, [r/p]\Gamma' \vdash [r/p]A$ *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 4.5 is the first of these, showing that the $(\xrightarrow{book})$ relation has the necessary property.

Lemma 4.5 *If $\Gamma \vdash e : \tau$ is derivable, and $e \stackrel{book}{\longrightarrow} e'$, then $\Gamma \vdash e' : \tau$ is derivable.*

The proof of Lemma 4.5 consists of two parts: the first shows that a derivation from $\Gamma \vdash e : \tau$ can only depend on the form of τ, not on the form of e. More formally, if $\Gamma \vdash C[e] : \tau$ is derived from $\Gamma' \vdash e : \sigma$ and $\Gamma' \vdash e' : \sigma$ is also derivable, then so is $\Gamma \vdash C[e'] : \tau$. This fact is easily seen by an inspection of the typing rules. The second part shows that if $\Gamma \vdash e : \tau$ is derivable, and $e \stackrel{book}{\longrightarrow} e'$ by e matching the left-hand side of one of the ($\stackrel{book}{\longrightarrow}$) axioms, then $\Gamma \vdash e' : \tau$ is also derivable. This fact follows from a case analysis of the four ($\stackrel{book}{\longrightarrow}$) axioms. Lemma 4.4 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 4.6 (Expression Substitution) *If $\Gamma, x : \tau_1, \Gamma' \vdash e_2 : \tau_2$ and $\Gamma \vdash e_1 : \tau_1$ are both derivable, then so is $\Gamma, \Gamma' \vdash [e_1/x]e_2 : \tau_2$.*

Lemma 4.6 is proved by induction on the derivation of $\Gamma, x : \tau_1, \Gamma' \vdash e_2 : \tau_2$.

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 4.7 (Method Bodies are Type Correct) *If $\Gamma \vdash_N \langle e_1 \leftarrow\!\!\ominus\, m = e_2 \rangle : \mathbf{pro}\, t . R$ 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 $\Gamma, t : \{t^+\} \vdash_N R <:_w \langle\!\langle m : \tau \rangle\!\rangle$ and $\Gamma \vdash_N e_2 : [\mathbf{pro}\, t . R/t](t \rightarrow \tau)$ are both derivable. Furthermore, if $\leftarrow\!\!\ominus = \leftarrow$ then $\Gamma \vdash_N e_1 : \mathbf{pro}\, t . R$ is also derivable.*

Lemma 4.8 *If $\Gamma, t : \{t^+\}, \Gamma_\tau \vdash_N U_1 <:_B U_2$, $\Gamma \vdash_N U_2 : -\gamma_2$, $\Gamma \vdash_N \tau_1 <: \tau_2$, and $\Gamma \vdash_N \tau_i : V_i$ for $i \in \{1, 2\}$ are all derivable, then*

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

where Γ_τ is a context listing only type variables.

Lemma 4.9 (Object Types Come From Pro Types) *If $\Gamma \vdash_N \langle e_1 \leftarrow\!\!\ominus\, m = e_2 \rangle : \mathbf{obj}\, t . R$ is derivable, then there exists a type $\mathbf{pro}\, t . R'$ such that*

$$\Gamma \vdash_N \langle e_1 \leftarrow\!\!\ominus\, m = e_2 \rangle : \mathbf{pro}\, t . R'$$
$$\Gamma \vdash_N \mathbf{pro}\, t . R' <: \mathbf{obj}\, t . R$$

are both derivable.

Theorem 4.10 (Subject Reduction) *If* $\Gamma \vdash e : \tau$ *is derivable, and* $e \xrightarrow{eval} e'$, *then* $\Gamma \vdash e' : \tau$ *is also derivable.*

The proof is similar in outline to that of Lemma 4.5; 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 4.5, the (β) case from Lemma 4.6, the ($\Leftarrow$) case from Lemmas 4.7, 4.8, and 4.9, and the ($Class$) case from Lemmas 4.1 and 4.6.

Theorem 4.11 (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.*

5 Analysis

While space limitations prevent a thorough discussion, we briefly note several implications of our approach.

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 [**?**, MM85, Mit86, Mit91] give various sufficient conditions on interchangeability of implementations.

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, **?**] 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.

We can also see from our approach that there are problems with C++ `private virtual` functions. Specifically, it is necessary to include "negative information" in the interface of an abstract type if we want to be able to extend the representation. 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 anomolous 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. The model uses standard notions from type theory, specifically existential types with higher-order bounded quantification, to model encapsulation; objects and inheritance are provided by a calculus extending previous work [Mit90, FHM94, FM95]. We have devised a type soundness proof for our calculus based on a straightforward operational semantics.

Write new short conclusion saying:
the main point of the paper
conclusions from the analysis (referring to Section 5)

Although space limitations prevent us from exploring this idea in detail, we can also account for *friend* functions by essentially the same mechanism.

State directions for further work:
C++ "abstract" classes (which may just be what we have, with kind saying that implementions are not provided ???)
similarly, partially abstract classes
clean up special syntax (if we don't get it done)
reformulate using modules and dot notation [**?**, HL91]

Also binding private names so they can't escape and removing the necessity of the negative kind information in the abstype declaration.

——————————————

(** is the first part of the following paragraph correct ?? put in analysis section ??

This issue is related to the connections between protection levels, binary methods and subtyping. Private binary methods do not break subtyping (because they can only be passed objects of the same class). Public binary methods do break subtyping. For protected binary methods, the issue is more subtle. It hinges on exactly the issue described below. If we may access the protected methods of our parents (and thus our siblings, via subsumption), then protected binary methods *do* break subtyping. If we can only access the protected methods from the self object, then protected binary methods do not break subtyping. Since protection in C++ is at the class level anyway, it probably would be better to allow the code below to work. Their rationale for disallowing it was something like you shouldn't be allowed to call sibling code. It's in the ARM somewhere. I can track it down if you are interested.

In our system, protection is at the "class" level, so protected methods do break subtyping. Actually, in the current formulation, private binary methods break subtyping. This is something I should eventually fix.

I thought through these issues a while ago, but I don't remember all the details now. I can look up my notes and track down the exact details if you think it's worth it.

**)

A problem in the C++ object system arises in connection with protected members of base classes. In C++, it is possible for a derived class member function to access protected members of a base class. For example, if B is a public base class of C, and B declares protected member function f, then a C object o can call member function f. However, if another B object, o', is passed as a parameter to a member function of o, it is not possible for the member function to call $o'.f$. We can see no semantic reason for this restriction. Moreover, if it were possible to access base class protected members of parameters, this would proved a very useful type-safe method for sibling classes to share non-public information. A specific use could be to have an abstract base class of sets with a union operation, but no specified internal representation of sets. Two derived classes could have different internal representations of sets, say bit vectors and linked lists, with a base class protected member function used to convert to a third "intermediate" representation. With the union member function of each derived class allowed to call this protected member, we could guarantee interoperability between the two classes without making the interconversion function public.

References

[AC94] M. Abadi and L. Cardelli. A theory of primitive objects: untyped and first-order systems. In *Proc. Theor. Aspects of Computer Software*, Sendai, Japan, 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.

[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 *Proc. 10th Int'l Conf. Fundamentals of Computation Theory (FCT'95)*, pages 42–61. Springer LNCS 965, 1995.

[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.

[HL91] R. Harper and M. Lillibridge. A bug in sml polymorphic typing for continuations. Email to `types@theory.lcs.mit.edu`, 1991.

[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.

[Mit90] J.C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Proc. 17th ACM Symp. on Principles of Programming Languages*, pages 109–124, January 1990.

[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.

[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:

(*switch ext ov*)	$\langle\langle \mathtt{e}_1 \leftarrow \mathtt{l}_2{=}\mathtt{e}_2 \rangle \leftarrow\!\!+\, \mathtt{l}_3{=}\mathtt{e}_3 \rangle$	$\xrightarrow{book}$	$\langle\langle \mathtt{e}_1 \leftarrow\!\!+\, \mathtt{l}_3{=}\mathtt{e}_3 \rangle \leftarrow \mathtt{l}_2{=}\mathtt{e}_2 \rangle$
(*perm ov ov*)	$\langle\langle \mathtt{e}_1 \leftarrow \mathtt{l}_2{=}\mathtt{e}_2 \rangle \leftarrow \mathtt{l}_3{=}\mathtt{e}_3 \rangle$	$\xrightarrow{book}$	$\langle\langle \mathtt{e}_1 \leftarrow \mathtt{l}_3{=}\mathtt{e}_3 \rangle \leftarrow \mathtt{l}_2{=}\mathtt{e}_2 \rangle$
(*add ov*)	$\langle \mathtt{e}_1 \leftarrow\!\!+\, \mathtt{l}_2{=}\mathtt{e}_2 \rangle$	$\xrightarrow{book}$	$\langle\langle \mathtt{e}_1 \leftarrow\!\!+\, \mathtt{l}_2{=}\mathtt{e}_2 \rangle \leftarrow \mathtt{l}_2{=}\mathtt{e}_2 \rangle$
(*cancel ov ov*)	$\langle\langle \mathtt{e}_1 \leftarrow \mathtt{l}_3{=}\mathtt{e}_2 \rangle \leftarrow \mathtt{l}_3{=}\mathtt{e}_3 \rangle$	$\xrightarrow{book}$	$\langle \mathtt{e}_1 \leftarrow \mathtt{l}_3{=}\mathtt{e}_3 \rangle$
(β)	$(\lambda \mathtt{x}.\, \mathtt{e}_1)\mathtt{e}_2$	$\xrightarrow{eval}$	$[\mathtt{e}_2/\mathtt{x}]\mathtt{e}_1$
($\Leftarrow$)	$\langle \mathtt{e}_1 \leftarrow\!\ominus\, \mathtt{l}{=}\mathtt{e}_2 \rangle \Leftarrow \mathtt{l}$	$\xrightarrow{eval}$	$\mathtt{e}_2 \langle \mathtt{e}_1 \leftarrow\!\ominus\, \mathtt{l}{=}\mathtt{e}_2 \rangle$ where $\leftarrow\!\ominus$ may either $\leftarrow\!\!+$ or $\leftarrow$.
(*Abstype*)	$\mathtt{Abstype}\ \mathtt{r} <:_{\mathtt{w}} \mathtt{R}\ ::\kappa\ \mathtt{with}\ \mathtt{x} : \tau$ $\mathtt{is}\ \{\!|\, \mathtt{r} <:_{\mathtt{w}} \mathtt{R}\ ::\kappa = \mathtt{R}',\ \mathtt{e}_1 \,|\!\}\ \mathtt{in}\ \mathtt{e}_2$	$\xrightarrow{eval}$	$[\mathtt{R}'/\mathtt{r}, \mathtt{e}_1/\mathtt{x}]\mathtt{e}_2$

B Type System

Types

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

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

$$kind ::= V \mid \kappa$$
$$\kappa ::= T^a \rightarrow \nu \mid \nu, \qquad a ::= + \mid - \mid o \mid ?$$
$$\nu ::= (M; V)$$
$$M ::= \{\vec{m}\}$$
$$V ::= \{\vec{t^b}\},\ b ::= + \mid - \mid o$$

Contexts

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

The judgement forms are:

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

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

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

$(start)$

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

$(type\ var)$

$$
\frac{\Gamma \vdash * \qquad t \notin dom(\Gamma)}{\Gamma,\, t : \{t^+\} \vdash *}
$$

$(row\ var)$

$$
\frac{\Gamma \vdash R_1 :: S_1 \to (M_1;\, V_1) \qquad S_0 \to (M_0;\, V_0) \leq S_1 \to (M_1;\, V_1) \qquad D(V_0) \subseteq dom(\Gamma) \qquad r \notin dom(\Gamma)}{\Gamma,\, (r <:_w R_1 :: S_0 \to (M_0;\, V_0)) \vdash *}
$$

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

$$(weakening) \qquad \frac{\begin{array}{c}\Gamma_1,\, \Gamma_2 \vdash A \\ \Gamma_1, a,\, \Gamma_2 \vdash *\end{array}}{\Gamma_1, a,\, \Gamma_2 \vdash A}$$

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

C.2 Rules for type expressions

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

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

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

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

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

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

C.3 Rules for rows

$(row\ proj)$

$$\frac{\Gamma \vdash * \qquad r <:_w R :: \kappa \in dom(\Gamma)}{\Gamma \vdash r :: \kappa}$$

$(empty\ row)$

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

$(row\ label)$

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

$(row\ fn\ abs)$

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

$(row\ fn\ app\ cov)$

$$\frac{\Gamma \vdash R :: T^+ \to (M; V_1) \qquad \Gamma \vdash \tau : V_2}{\Gamma \vdash R\tau :: (M; Merge(V_1, V_2))}$$

$(row\ fn\ app\ contra)$

$$\frac{\Gamma \vdash R :: T^- \to (M; V_1) \qquad \Gamma \vdash \tau : V_2}{\Gamma \vdash R\tau :: (M; Merge(V_1, \overline{V_2}))}$$

$(row\ fn\ app\ inv)$

$$\frac{\Gamma \vdash R :: T^o \to (M; V_1) \qquad \Gamma \vdash \tau : V_2}{\Gamma \vdash R\tau :: (M; Merge(V_1, Invar(V_2)))}$$

$(row\ fn\ app\ vac)$

$$\frac{\Gamma \vdash R :: T^? \to (M; V_1) \qquad \Gamma \vdash \tau : V_2}{\Gamma \vdash R\tau :: (M; V_1)}$$

$(row\ ext)$

$$\frac{\Gamma \vdash R :: (\{\vec{m}, m\}, V_1) \qquad \Gamma \vdash \tau : V_2}{\Gamma \vdash \langle\!\langle R \mid m : \tau \rangle\!\rangle :: (\{\vec{m}\}; Merge(V_1, V_2))}$$

C.4 Subtyping rules for types

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

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

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

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

C.5 Subtyping Rules for rows

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

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

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

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

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

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

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

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

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

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

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

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 $\rightarrow_\beta$, or β-conversion, written $\leftrightarrow_\beta$.

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

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

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

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

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

C.6 Rules for assigning types to terms

$(exp\ proj)$
$$\frac{\Gamma \vdash * \qquad x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

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

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

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

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

$(pro\ ext)$
$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \mathbf{pro}\, t.R \\ \Gamma,\ t : \{t^+\} \vdash R :: (\{m\}; V) \\ \Gamma,\ I_r \vdash e_2 : [\mathbf{pro}\, t.rt/t](t \rightarrow \tau) \qquad r \notin V(\tau)\end{array}}{\Gamma \vdash \langle e_1 \leftarrow\!\!+\ m{=}e_2\rangle : \mathbf{pro}\, t.\langle\!\langle R \mid m : \tau\rangle\!\rangle}$$

where $I_r = r <:_w \lambda t.\langle\!\langle R \,|\, m : \tau \rangle\!\rangle :: T^0 \rightarrow (\emptyset;\, Invar(V_\Gamma))$.

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

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

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

$$(\exists <:\ intro) \qquad \frac{\begin{array}{c}\Gamma \vdash R_1 :: S_1 \rightarrow (M_1;\, V_1) \\ \Gamma \vdash R_1 <:_w R \\ \Gamma \vdash e : [R_1/r]\tau\end{array}}{\Gamma \vdash \{\!|\, r <:_w R :: \kappa = R_1,\ e \,|\!\} : \exists (r <:_w R :: \kappa)\tau}$$

where $\kappa = S_1 \rightarrow (M_1;\, V_1)$.

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