Extending Moby with inheritance-based subtyping

Kathleen Fisher[1] and John Reppy[2]

[1] AT&T Labs — Research
180 Park Avenue, Florham Park, NJ 07932 USA
kfisher@research.att.com
[2] Bell Laboratories, Lucent Technologies
700 Mountain Avenue, Murray Hill, NJ 07974 USA
jhr@research.bell-labs.com

Abstract. Classes play a dual role in mainstream statically-typed object-oriented languages, serving as both object generators and object types. In such languages, inheritance implies subtyping. In contrast, the theoretical language community has viewed this linkage as a mistake and has focused on subtyping relationships determined by the structure of object types, without regard to their underlying implementations. In this paper, we explore why inheritance-based subtyping relations are useful and we present an extension to the MOBY programming language that supports both inheritance-based and structural subtyping relations. In addition, we present a formal accounting of this extension.

1 Introduction

There is a great divide between the study of the foundations of object-oriented languages and the practice of mainstream object-oriented languages like JAVA[AG98] and C++[Str97]. One of the most striking examples of this divide is the rôle that class inheritance plays in defining subtyping relations. In most foundational descriptions of OO languages, and in the language designs that these studies have informed, inheritance does not define any subtyping relation, whereas in languages like JAVA and C++, inheritance defines a subtyping hierarchy. What is interesting about this distinction is that there are certain idioms, such as friend functions and binary methods, that are natural to express in an inheritance-based subtyping framework, but which require substantial complication to handle in a structural subtyping framework. It is important not to confuse inheritance-based subtyping with *by-name* subtyping, where the subtyping relationships between object types are explicitly declared (*e.g.*, JAVA's interfaces). While both inheritance-based and by-name subtyping avoid the *accidental subtyping* problem that structural subtyping suffers from,[1] the type names in an inheritance-based scheme are tied to specific implementations, whereas multiple, unrelated classes may be declared to implement the same type name in a by-name scheme.

In this paper, we explore why inheritance-based subtyping relations are useful, and we present an extension to the MOBY programming language [FR99a] that supports such subtyping relations by adding *class types*. While inheritance-based subtyping can

[1] The common example is a cowboy and widget that both have draw methods.

be found in JAVA, C++, and other languages, the approach we take in Extended MOBY has several novel aspects. The most significant of these is that class types can be in an inheritance-based subtyping relationship even when their corresponding object types are not in the corresponding structural subtyping relationship. We also present a formal accounting of an object calculus that models the typing issues presented by Extended MOBY.

We begin by examining the common object-oriented idiom of *friend functions* and exploring how one might implement this idiom in MOBY [FR99a], which is a language with only structural subtyping. This example illustrates the deficiency of relying solely on structural subtyping in the language design. We then describe Extended MOBY and class types in Section 3. We show how this extension supports a number of common idioms, such as friend functions, binary methods, and object cloning. In Section 4, we present XMOC, an object calculus that models the type system of Extended MOBY. Specifically, XMOC supports both structural and inheritance-based subtyping, as well as privacy. To validate the design of Extended MOBY, we prove the type soundness of XMOC. In Section 5 we describe related work and we conclude in Section 6.

2 The problem with friends

Both C++ and JAVA have mechanisms that allow some classes and functions to have greater access privileges to a class's members than others. In C++, a class grants this access by declaring that certain other classes and functions are *friends*. In JAVA, members that are not annotated as **`public`**, **`protected`**, or **`private`** are visible to other classes in the same package, but not to those outside the package. In this section, we examine how to support this idiom in MOBY, which has only structural subtyping. This study demonstrates that while it is possible to encode the friends idiom in a language with only structural subtyping, the resulting encoding is not very appealing.

MOBY is a language that combines an ML-style module system with classes and objects [FR99a]. Object types are related by structural subtyping, which is extended to other types in the standard way. One of MOBY's most important features is that it provides flexible control over class-member visibility using a combination of the class and module mechanisms. The class mechanism provides a visibility distinction between clients that use the class to create objects and clients that use the class to derive subclasses, while the module mechanism provides hiding of class members via signature matching. The names of class members that are hidden by signature matching are truly private and can be redefined in a subclass. A brief introduction to MOBY is given in Appendix A.

2.1 Friends via partial type abstraction

A standard way to program friends is to use partially abstract types [PT93,KLM94]. For example, Figure 1 gives the MOBY code for an implementation of a `Bag` class that has a `union` function as a friend. In this example, we have ascribed the `BagM` module with a signature that makes the `Rep` type partially abstract to the module's clients. Outside the module, if we have an object of type `Rep`, we can use both the `union` function

```
module BagM : {
  type Rep <: Bag
  objtype Bag { meth add : Int -> Unit }
  val union : (Rep, Rep) -> Unit
  val mkBag : Unit -> Rep
} {
  class Bag {
    public field items : var List(Int)
    public meth add (x : Int) -> Unit {
      self.items := x :: self.items
    }
    public maker mk () { field items = Nil }
  }
  objtype Rep = typeof(Bag)
  objtype Bag { meth add : Int -> Unit }
  fun union (s1 : Rep, s2 : Rep) -> Unit {
    List.app s1.add s2.items
  }
  fun mkBag () -> Rep = new mk()
}
```

Fig. 1. Bags and friends using type abstraction

and the `add` method (since `Rep` is a subtype of `Bag`), but we cannot access the `items` field. Inside the module, the `Rep` type allows access to all of the members of the `Bag` class;[2] the implementation of the `union` function exploits this access. Note that the `items` field is **`public`** inside the `BagM` module, but is not part of `BagM`'s interface outside the module. Thus, objects created from subclasses of `Bag` are not known to be structural subtypes of `Rep`.

Unfortunately, this approach only works for *final* classes. If we want to extend the `Bag` class, we must reveal the class in the signature of the `BagM` module (as is done in Figure 2). In this version, an object created using the `mk` maker cannot be used as an argument to the `union` function. This limitation also applies to objects created from subclasses of `Bag`.

2.2 Friends via representation methods

To support friends and class extension in the same class requires a public mechanism for mapping from an object to its abstract representation type. With this mechanism, we can recover the representation type required by the friend functions. For example, suppose we extend our `Bag` class to include a method that returns the number of items in the bag. We call this new class `CBag` (for counting bag), and we want to use the `union` function on objects created from the `CBag` class. Figure 3 presents this new implementation. Notice that we have added a public method `bagRep` to the interface

[2] The MOBY notation **`typeof`**`(`*C*`)` is shorthand for the object type that consists of the public members of class *C*.

```
module BagM : {
  type Rep <: typeof(Bag)
  class Bag {
    public meth add : Int -> Unit
    public maker mk of Unit
  }
  val union : (Rep, Rep) -> Unit
  val mkBag : Unit -> Rep
} {
  ...
}
```

Fig. 2. Revealing the `Bag` class

of the `Bag` class, which returns **`self`** at the representation type (`Rep`). To apply the `union` function to two bags `b1` and `b2`, we write "`Bag.union (b1.bagRep(), b2.bagRep())`." This expression works even when `b1` and/or `b2` are counting bags.

Although this example does not include friends for the `CBag` class, we have included the representation method in its interface, which illustrates the main weakness of this approach. Namely, for each level in the class hierarchy, we must add representation types and methods. These methods pollute the method namespace and, in effect, partially encode the class hierarchy in the object types. Furthermore, it suffers from the source-code version of the *fragile base-class* problem: if we refactor the class hierarchy to add a new intermediate class, we have to add a new representation method, which changes the types of the objects created below that point in the hierarchy. While this encoding approach appears to be adequate for most of the examples that require a strong connection between the implementation and types, it is awkward and unpleasant.

3 Extended MOBY

In the previous section, we showed how we can use abstract representation types and representation methods to tie object types to specific classes. From the programmer's perspective, a more natural approach is to make the classes themselves serve the rôle of types when this connection is needed. In this section, we present an extension of MOBY [FR99a] that supports such *class types* and *inheritance-based subtyping*. Intuitively, an object has a class type **#**`C` if the object was instantiated from `C` or one of its descendants. Inheritance-based subtyping is a form of by-name subtyping that follows the inheritance hierarchy. We illustrate this extension using several examples.

3.1 Adding inheritance-based subtyping

Inheritance-based subtyping requires four additions to MOBY's type system, as well as a couple of changes to the existing rules:

- For any class `C`, we define **#**`C` to be its *class type*, which can be used as a type in any context that is in `C`'s scope. Note that the meaning of a class type depends on

```
module BagM : {
  type Rep <: typeof(Bag)
  class Bag : {
    public meth add : Int -> Unit
    public meth bagRep : Unit -> Rep
    public maker mkBag of Unit
  }
  val union : (Rep, Rep) -> Unit
} {
  class Bag {
    public field items : var List(Int)
    public meth add (x : Int) -> Unit {
      self.items := x :: self.items
    }
    public meth bagRep () -> Rep { self }
    public maker mkBag () { field items = Nil }
  }
  objtype Rep = typeof(Bag)
  fun union (s1 : Rep, s2 : Rep) -> Unit {
    List.app s1.add s2.items
  }
}

module CBagM : {
  type Rep <: typeof(CBag)
  class CBag : {
    public meth add : Int -> Unit
    public meth bagRep : Unit -> BagM.Rep
    public meth size : Unit -> Int
    public meth cbagRep : Unit -> Rep
    public maker mkCBag of Unit
  }
} {
  class CBag {
    inherits BagM.Bag
    public field nItems : var Int
    public meth add (x : Int) -> Unit {
      self.nItems := self.nItems+1;
      super.add(x)
    }
    public meth size () -> Int { self.nItems }
    public meth cbagRep () -> Rep { self }
    public maker mkCBag () { super mkBag(); field nItems = 0 }
  }
  objtype Rep = typeof(CBag)
}
```

Fig. 3. Bags and friends using representation methods

```
class B {
  public meth m1 () -> Int { ... }
  public meth m2 ...
  ...
}
class C {
  inherits B : { public meth m2  ... }
  public meth m1 () -> Bool { ... }
  maker mkC of Unit { ... }
  ...
}
```

Fig. 4. Example of reusing a private method name

its context. Inside a method body, the class type of the host class allows access to all members, whereas outside the class, only the public members can be accessed.

– We extend class interfaces to allow an optional **inherits** clause. If in a given context, a class C has an interface that includes an "**inherits** B" clause, then we view **#**C as a subtype of **#**B. Omitting the **inherits** clause from C's interface causes the relationship between B and C to be hidden.
– We say that **#**C is a subtype of **typeof**(C) (this relation corresponds to Fisher's observation that implementation types are subtypes of interface types [Fis96]).
– The existing typing judgements for method and field selection require the argument to have an object type. We add new judgements for the case where the argument has a class type. We add new rules, instead of adding subsumption to the existing rules, to avoid a technical problem that is described in Section 3.2.
– When typing the methods of a class C, we give **self** the type **#**C (likewise, if B is C's superclass, then **super** has the type **#**B).
– When typing a **new** expression, we assign the corresponding class type to the result.

3.2 Inheritance-based subtyping vs. privacy

There is a potential problem in the Extended MOBY type system involving the interaction between inheritance-based subtyping and MOBY's support for privacy. Because MOBY allows signature ascription to hide object members (*e.g.*, the items field in Figure 2), **#**C can be a subtype of **#**B even when **typeof**(C) is not a subtype of **typeof**(B). The problem arises in the case where class C has defined a method that has the same name as one of B's private methods. Consider the code fragment in Figure 4, for example.[3] Given these definitions, how do we typecheck the expression: "(**new** mkC()).m1()?" If we allow subsumption on the left-hand side of the method selection, then there are two incompatible ways to typecheck this expression. To avoid this ambiguity, we disallow subsumption on the left-hand side of member selection, and instead have two different rules for typing member selection: one for the

[3] This example uses a *class interface* annotation on the class B; this syntactic form avoids the need to wrap B in a module and signature to hide the m2 method.

case where the object's type is a class type and one for the case where the object's type is an object type. An alternative design is to restrict the inheritance-based subtyping to hold between **#**`C` and **#**`B` only when **`typeof`**`(C)` is a subtype of **`typeof`**`(B)`. The downside is that such a restriction reduces the power of the mechanism; for example, the mixin encoding described in Section 3.6 would not work.

3.3 Friends revisited

We can now revisit our bag class example using the inheritance-based subtyping features of Extended MOBY. In this new implementation (see Figure 5), we use the class type **#**`Bag` instead of the `Rep` type, which allows us to simplify the code by both eliminating the `Rep` type and the representation method. Note that the interface for the `CBag` class includes an **`inherits`** clause that specifies that it is a subclass of `Bag`. This relation allows the `union` function to be used on values that have the **#**`CBag` type.

3.4 Binary methods

Binary methods are methods that take another object of the same class as an argument [BCC$^+$96]. There are a number of different flavors of binary methods, depending on how objects from subclasses are treated. Using class types, we can implement binary methods that require access to the private fields of their argument objects. For example, Figure 6 shows how the `union` function in the previous example can be implemented as a binary method.

3.5 Object cloning

Another case where inheritance-based subtyping is useful is in the typing of *copy constructors*, which can be used to implement a user-defined object cloning mechanism.[4] Figure 7 gives an example of cloning in Extended MOBY. Class `B` has a private field (`pvtX`), which makes object types insufficient to type check `C`'s use of the `copyB` maker function. The problem arises because the object type associated with **`self`** in type-checking `C` does not have a `pvtX` field (because that field is private to `B`), but the `copyB` maker function requires one. Thus, we need the inheritance-based subtyping relationship to allow the `copyC` maker to pass **`self`**, typed with **#**`C`, as a parameter to the `copyB` maker. Because we know that `C` inherits from `B`, this application typechecks. We also exploit this subtyping relation when we override the `clone` method.

3.6 Encoding mixins

MOBY does not support any form of multiple inheritance, but with the combination of parameterized modules and class types, it is possible to encode mixins [BC90,FKF98]. In this encoding, a mixin is implemented as a class parameterized over its base class using a parameterized module. The class interface of the base class contains only those components that are necessary for the mixin. After applying the mixin to a particular

[4] Note that in MOBY, constructors are called *makers*.

```
module BagM : {
  class Bag : {
    public meth add : Int -> Unit
    public maker mkBag of Unit
  }
  val union : (#Bag, #Bag) -> Unit
} {
  class Bag {
    public field items : var List(Int)
    public meth add (x : Int) -> Unit {
      self.items := x :: self.items
    }
    public maker mkBag () { field items = Nil }
  }
  fun union (s1 : #Bag, s2 : #Bag) -> Unit {
    List.app s1.add s2.items
  }
}

module CBagM : {
  class CBag : {
    inherits BagM.Bag
    public meth size : Unit -> Int
    public maker mkCBag of Unit
  }
} {
  class CBag {
    inherits BagM.Bag
    public field nItems : var Int
    public meth add (x : Int) -> Unit {
      self.nItems := self.nItems+1;
      super.add(x)
    }
    public meth size () -> Int { self.nItems }
    public maker mkCBag () { super mkBag(); field nItems = 0 }
  }
}
```

Fig. 5. Bags with friends in Extended MOBY

base class, we create a new class that inherits from the mixed base class and uses the class types to reconstitute the methods of the base class that were hidden as a result of the module application. Without class types, it would not be possible to make the original class's methods visible again. For example, Figure 8 gives the encoding of a mixin class that adds a `print` method to a class that has a `show` method. After applying `PrintMix` to class `A`, we define a class `PrA` that reconstitutes `A`'s `anotherMeth` method. Notice that we need to use an explicit type constraint to convert the type of **`self`** from **#**`PrA` to **#**`A`, since we do not have subsumption at method dispatch.

```
class Bag {
  field items : var List(Int)
  public meth add (x : Int) -> Unit {
    self.items := x :: self.items
  }
  public meth union (s : #Bag) -> Unit {
    List.app self.add s.items
  }
  public maker mkBag () { field items = Nil }
}
```

Fig. 6. Bags with a binary `union` method.

```
class B : {
  public meth getX : Unit -> Int
  public meth clone : Unit -> #B
  public maker mkB of Int
  maker copyB of #B
} {
  public meth getX () -> Int { self.pvtX }
  public meth clone () -> #B { new copyB(self) }
  public maker mkB (x : Int) { field pvtX = x }

  field pvtX : Int
  maker copyB (orig : #B) { field pvtX = orig.pvtX }
}

class C {
  inherits B
  public meth clone () -> #C { new copyC(self) }
  public maker mkC (y : Int) { super mkB(y) }
  maker copyC (orig : #C) { super copyB(orig) }
}
```

Fig. 7. Cloning with privacy in Extended MOBY

While this encoding is cumbersome, it illustrates the power of class types. Also, it might serve as the definition of a derived form that directly supports mixins.

3.7 Efficiency of method dispatch

Although it is not our main motivation, it is worth noting that method dispatch and field selection from an object with a class type can be implemented easily as a constant time operation. When the dispatched method is final in the class type, the compiler can eliminate the dispatch altogether and call the method directly. In contrast, when an object has an object type, the compiler knows nothing about the layout of the object, making access more expensive.

```
signature HAS_SHOW {
  type InitB
  class B : {
    meth show : Unit -> String
    maker mk of InitB
  }
}
module PrintMix (M : HAS_SHOW)
{
  class Pr {
    inherits M.B
    public meth print () -> Unit {
      ConsoleIO.print(self.show())
    }
    maker mk (x : InitB) { super mk(x) }
  }
}

class A {
  public meth show () -> String { "Hi" }
  public meth anotherMeth () -> Unit { ... }
  maker mk () { }
}

module P = PrintMix({type InitB = Unit; class B = A})

class PrA {
  inherits P.Pr
  public meth anotherMeth () -> Unit {
    (self : #A).anotherMeth()
  }
}
```

Fig. 8. Encoding mixins in Extended MOBY

4 XMOC

We have developed a functional object calculus, called XMOC, that models the type system of Extended MOBY and validates its design. XMOC supports both traditional structural subtyping and inheritance-based subtyping. In this section, we discuss the intuitions behind XMOC and state type soundness results; space considerations preclude a more detailed presentation. The full system is given in Appendices B and C.

4.1 Syntax

The term syntax of XMOC is given in Figure 9. An XMOC program consists of a sequence of class declarations terminated by an expression. Each declaration binds a

$$
\begin{array}{rcl}
p & ::= & d;\ p \\
  & | & e \\[4pt]
d & ::= & \texttt{class}\ C\ (x : \tau)\ \{\!|\ b\ \ m = \mu_m{}^{m \in \mathcal{M}}\ |\!\} \\
  & | & \texttt{class}\ C : \sigma = C' \\[4pt]
b & ::= & \\
  & | & \texttt{inherits}\ C(e); \\[4pt]
\sigma & ::= & (\tau)\ \{\!|\ \texttt{inherits}\ B;\ \ m : \tau_m{}^{m \in \mathcal{M}}\ |\!\} \\[4pt]
\tau & ::= & \alpha \\
  & | & \tau \to \tau' \\
  & | & \texttt{obj}\ \alpha\,\texttt{.}\ \{\!|\ m : \tau_m{}^{m \in \mathcal{M}}\ |\!\} \\
  & | & \texttt{\#}C
\end{array}
\qquad
\begin{array}{rcl}
\mu & ::= & (x\ :\ \tau) \Rightarrow e \\[4pt]
e & ::= & x \\
  & | & \texttt{fn}(x\ :\ \tau) \Rightarrow e \\
  & | & e(e') \\
  & | & \texttt{new}\ C(e) \\
  & | & \texttt{self} \\
  & | & e\,\texttt{.}\,m \\
  & | & \texttt{self!state} \\
  & | & e\ @\ C
\end{array}
$$

Fig. 9. Syntax of XMOC terms

class name to a class definition; we do not allow rebinding of class names. The set of class names includes the distinguished name $\mathbf{None}$, which is used to denote the super-class of base classes. We follow the convention of using C for class names other than $\mathbf{None}$ and B for class names including $\mathbf{None}$. Class declarations come in two forms. In the first, a class C is defined by extending an optional parent class (b) by overriding and adding methods. When no parent is specified, we say that C is a *base-class*; otherwise, we say that C *inherits* from its parent class. To model the notion of object state, we parameterize this form of class declaration by a variable x. When an object is created, the argument bound to x is bound into the representation of the object as its state. In addition, x is used to compute the argument for the superclass. Since an object's implementation is spread over a hierarchy of classes, it has a piece of state for each class in its implementation. Methods of a given class may access the state for that class using the form **`self!state`**. In the second form of class declaration, a class C can be derived from an existing class C' by *class-interface ascription*, which produces a class that inherits its implementation from C', but has a more restrictive class interface σ. A class interface σ specifies the type of the class's state variable, the name of the nearest revealed ancestor class (or $\mathbf{None}$), and a typed list of available methods. The types of XMOC include type variables, function types, recursive object types, and class types.

Each method (μ) takes a single parameter and has an expression for its body. The syntax of expressions (e) includes variables, functions, function application, new object creation, the special variable **`self`** (only allowed inside method bodies), method dispatch, and access to the object's state. The last expression form ($e\ @\ C$) is an *object-view coercion*. Unlike Extended MOBY, XMOC does not map the inheritance relation directly to the subtyping relation; instead we rely on object-view coercions to explicitly coerce the type of an expression from a class to one of its superclasses. This approach avoids the problem discussed in Section 3.2 without requiring two typing judgements for method dispatch. It is possible to automatically insert these coercions into the XMOC representation of a program as part of typechecking (such a translation is similar to the type-directed representation wrapping that has been done for polymorphic languages [Ler92]).

4.2 Dynamic Semantics

Evaluation of an XMOC program occurs in two phases. The first phase is defined by the *class linking* relation, written $\mathcal{K}, p \rightarrowtail \mathcal{K}', p'$, which takes a *dynamic class environment* $\mathcal{K}$ and links the left-most class definition in p to produce $\mathcal{K}'$. Class linking terminates with a residual expression once all of the class declarations have been linked. The second phase evaluates the residual expression to a value (assuming termination). This phase is defined by the expression evaluation relation, which we write as $\mathcal{K} \vdash e \hookrightarrow e'$. Defining the semantics of linking and evaluation requires extending the term syntax with run-time forms.

Correctly handling class-interface ascription provides the greatest challenge in defining the semantics for XMOC. Using this mechanism, a public method m in B can be made private in a subclass C, and subsequently m can be reused to name an unrelated method in some descendant class of C (recall the example in Figure 4). Methods inherited from B must invoke the original m method when they send the m message to **`self`**, while methods defined in D must get the new version. We use a variation of the Riecke-Stone dictionary technique [RS98,FR99b] to solve this problem. Intuitively, dictionaries provide the α-conversion needed to avoid capture by mapping method names to *slots* in method tables. To adapt this technique to XMOC, when we link a class C, we replace each occurrence of **`self`** in the methods that C defines with the object view expression "$\mathit{self} @ C$." Rule 5 in Appendix B describes this annotation formally. At runtime, we represent each object as a pair of a raw object (denoted by meta-variable obj) and a view (denoted by a class name). The raw object consists of the object's state and the name of the defining class; this class name is used to find the object's method suite. The view represents the visibility context in which the message send occurs; those methods in scope in class C are available. To lookup method m in runtime object $\langle \mathit{obj},\ C \rangle$, we use the dictionary associated with C in the class environment $\mathcal{K}$ to find the relevant slot. We use this slot to index into the method table associated with obj. Rule 3 formally specifies method lookup.

4.3 Static semantics

The XMOC typing judgements are written with respect to a static environment Γ, which consists of a set of bound type variables ($\mathcal{A}$), a subtype assumption map ($\mathcal{S}$), a class environment ($\mathcal{C}$), and a variable environment ($\mathcal{V}$). We define these environments and give the complete set of XMOC typing judgements in Appendix C. Here we briefly discuss some of the more important rules.

As mentioned earlier, each XMOC class name doubles as an object type. We associate such a type with an object whenever we instantiate an object from a class, according to the typing rule

$$\frac{(\mathcal{C} \text{ of } \Gamma)(C) = (\tau) \{\!| \, \mathbf{inherits}\, B;\ m : {\tau_m}^{m \in \mathcal{M}} \,|\!\} \quad \Gamma \vdash e \triangleright \tau' \quad \Gamma \vdash \tau' <: \tau}{\Gamma \vdash \mathbf{new}\ C(e) \triangleright \#C}$$

which looks up class C in Γ, infers a type τ' for the constructor argument e, and insures that this type is a subtype of the type of the class parameter τ.

In contexts that allow subsumption, we can treat a class type as an object type according to the following subtyping judgement:

$$\frac{\Gamma \vdash \mathbf{Ok} \quad (\mathcal{C}\,\mathrm{of}\,\Gamma)(C) = (\tau)\,\{\!\!|\, \mathtt{inherits}\, B;\ m : \tau_m{}^{m \in \mathcal{M}} \,|\!\!\}}{\Gamma \vdash \mathtt{\#}C <: \mathtt{obj}\,\alpha\,\mathtt{.}\,\{\!\!|\, m : \tau_m{}^{m \in \mathcal{M}} \,|\!\!\}}$$

This rule corresponds to the property that `#C` is a subtype of **`typeof(C)`** in Extended MOBY. Note that unlike Extended MOBY, we do not treat a class type $\mathtt{\#}C$ as being a subtype of its superclass type. Instead we use an object view constraint, which is typed as follows:

$$\frac{\Gamma \vdash e \rhd \mathtt{\#}C' \quad \mathcal{H}(\Gamma) \vdash C' \lessdot C}{\Gamma \vdash e\,@\,C \rhd \mathtt{\#}C}$$

The judgement $\mathcal{H}(\Gamma) \vdash C' \lessdot C$ states that C' inherits from C in the class hierarchy $\mathcal{H}(\Gamma)$ (the meta-function $\mathcal{H}$ projects the static class hierarchy from the environment Γ). Because we do not treat inheritance directly as subtyping in XMOC, we only need one rule for typing method dispatch.

$$\frac{\Gamma \vdash e \rhd \tau \quad \Gamma \vdash \tau <: \mathtt{obj}\,\alpha\,\mathtt{.}\,\{\!\!|\, m : \tau_m{}^{m \in \mathcal{M}} \,|\!\!\} \quad m \in \mathcal{M}}{\Gamma \vdash e\,\mathtt{.}\,m \rhd \tau_m[\alpha \mapsto \mathtt{obj}\,\alpha\,\mathtt{.}\,\{\!\!|\, m : \tau_m{}^{m \in \mathcal{M}} \,|\!\!\}]}$$

4.4 Soundness

We have proven the type soundness of XMOC and we outline this result here. We start with a *subject reduction* theorem, which states that the evaluation relation preserves the type of programs and class environments.

Theorem 1 If $\Gamma \vdash \mathcal{K}$, and for a program p we have $\Gamma, \mathcal{H}(\mathcal{K}) \vdash p \rhd \Gamma_f, \tau_f$ and $\mathcal{K}, p \rightarrowtail \mathcal{K}', p'$, then there exists an environment Γ' and a type τ', such that

- $\Gamma', \mathcal{H}(\mathcal{K}') \vdash p' \rhd \Gamma_f, \tau'$,
- $\Gamma' \vdash \tau' <: \tau_f$, and
- $\Gamma' \vdash \mathcal{K}'$

Furthermore, if p is an expression, then $\Gamma_f = \Gamma'$.

The proof has two steps: we show that linking preserves types and produces a well-typed class environment, and then we show that expression evaluation preserves the type of expressions.

The next step is a *progress* theorem, which states that well-typed programs do not get *stuck*.

Theorem 2 If $\Gamma \vdash \mathcal{K}$, and for a program p we have $\Gamma, \mathcal{H}(\mathcal{K}) \vdash p \rhd \Gamma_f, \tau_f$ with p and τ_f both closed, then either p is a value or there exists $\mathcal{K}'$ and p' such that $\mathcal{K}, p \rightarrowtail \mathcal{K}', p'$.

Here we use $\mathcal{H}$ to project the *dynamic* class hierarchy from the dynamic class environment $\mathcal{K}$.

Definition 1 We say that a program p *yields* $\mathcal{K}', p'$ if $\emptyset, p \rightarrowtail \mathcal{K}', p'$ and there does *not* exist $\mathcal{K}'', p''$ such that $\mathcal{K}', p' \rightarrowtail \mathcal{K}'', p''$.

Finally, we show the type soundness of XMOC.

Theorem 3 If $\emptyset \vdash p \triangleright \Gamma_f, \tau_f$ and p yields $\mathcal{K}', p'$, then p' is a value w, such that $\Gamma_f, \mathcal{H}(\mathcal{K}') \vdash w \triangleright \tau'$ with $\Gamma_f \vdash \tau' <: \tau_f$.

5 Related work

Our class types are motivated by the rôle that classes play in languages like C++ and JAVA. The main difference between Extended MOBY and the class types provided by these other languages is in the way that abstraction is supported. Extended MOBY allows partial hiding of inherited components using signature ascription, which means that **`typeof`**`(C)` may not be a subtype of **`typeof`**`(B)` even when `C` is known to inherit from `B` (see Section 3.2). JAVA does not support private inheritance, but C++ allows a class to *privately* inherit from another class, but in that case all of the base-class members are hidden and there is no subtyping relationship. Extended MOBY is more flexible, since it allows hiding to be done on a per-member basis. It also allows the class hierarchy to be hidden by omitting the **`inherits`** clause in class interfaces. In C++ and JAVA the full class hierarchy is manifest in the class types (except for C++'s private inheritance). Another point of difference is that Extended MOBY supports structural subtyping on object types; JAVA has object types (called interfaces), but subtyping is *by-name*. C++ does not have an independent notion of object type.

Fisher's Ph.D. dissertation [Fis96] is the earliest formalization of class types that we are aware of. In her work, each implementation is tagged with a row variable using a form of bounded existential rows. In our work, we adopt classes as a primitive notion and use the names of such classes in a fashion analogous to Fisher's row variables. A weakness of the earlier work is its treatment of private names; it provides no way to hide a method and then later add an unrelated method with the same name.

Our use of dictionaries to specify the semantics of method dispatch in the presence of privacy is adapted from the work of Riecke and Stone [RS98]. The main difference is that XMOC has an explicit notion of class and we introduce dictionaries as a side-effect of linking classes.

More recently, Igarashi *et al.* have described *Featherweight Java*, which is an object calculus designed to model the core features of JAVA's type system [IPW99]. Like our calculus, Featherweight Java has a notion of subtyping based on class inheritance. Our calculus is richer, however, in a number of ways. Our calculus models private members and narrowing of class interfaces. We also have a notion of structural subtyping and we relate the implementation and structural subtyping notions.

The notion of type identity based on implementation was present in the original definition of *Standard ML* in the form of *structure sharing* [MTH90]. The benefits of structure sharing were fairly limited and it was dropped in the 1997 revision of SML [MTHM97].

6 Conclusion

This paper presents an extension to MOBY that supports classes as types. We have illustrated the utility of this extension with a number of examples. We believe that Extended MOBY is the first design that incorporates types for objects that range from class types to structural object types.[5] While this flexibility goes beyond what is found in other languages, the most interesting aspect of the design is the interaction between privacy (specified via module signature matching) and inheritance-based subtyping. Because we allow the inherits relation between two class types to be visible even when their corresponding object types are not in a subtyping relationship, one can use class types to recover access to hidden methods. This property enables more flexible use of parameterized modules in combining class implementations, as illustrated in Section 3.6.

We have also developed a formal model of this extension and have proven type soundness for it. We are continuing to work on improving our formal treatment of class types and implementation-based inheritance. One minor issue is that XMOC requires that class names be unique in a program; this restriction can be avoided by introducing some mechanism, such as *stamps*, to distinguish top-level names (*e.g.*, see Leroy's approach to module system semantics [Ler96]). We would also like to generalize the rule that relates class types with object types (rule 36 in Appendix C) to allow positive occurrences of $\#C$ to be replaced by the object type's bound type variable. While we believe that this generalization is sound, we have not yet proven it.

References

[AG98] Arnold, K. and J. Gosling. *The Java Programming Language*. Addison-Wesley, Reading, MA, 2nd edition, 1998.

[BC90] Bracha, G. and W. Cook. Mixin-based inheritance. In *ECOOP'90*, October 1990, pp. 303–311.

[BCC+96] Bruce, K., L. Cardelli, G. Castagna, The Hopkins Object Group, G. Leavens, and B. Pierce. On binary methods. *TAPOS*, **1**(3), 1996, pp. 221–242.

[Fis96] Fisher, K. *Type Systems for Object-oriented Programming Languages*. Ph.D. dissertation, Stanford University, August 1996.

[FKF98] Flatt, M., S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *POPL'98*, January 1998, pp. 171–183.

[FR99a] Fisher, K. and J. Reppy. The design of a class mechanism for Moby. In *PLDI'99*, May 1999, pp. 37–49.

[FR99b] Fisher, K. and J. Reppy. Foundations for MOBY classes. *Technical Memorandum*, Bell Labs, Lucent Technologies, Murray Hill, NJ, February 1999.

[IPW99] Igarashi, A., B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *OOPSLA'99*, November 1999, pp. 132–146.

[KLM94] Katiyar, D., D. Luckham, and J. Mitchell. A type system for prototyping languages. In *POPL'94*, January 1994, pp. 138–161.

[Ler92] Leroy, X. Unboxed objects and polymorphic typing. In *POPL'92*, January 1992, pp. 177–188.

[5] JAVA is close to Extended Moby in this respect, but interface subtyping relations in JAVA must be declared ahead of time by the programmer, whereas object-type subtyping in MOBY is based on the structure of the types.

[Ler96] Leroy, X. A syntactic theory of type generativity and sharing. *JFP*, **6**(5), September 1996, pp. 1–32.
[MTH90] Milner, R., M. Tofte, and R. Harper. *The Definition of Standard ML*. The MIT Press, Cambridge, MA, 1990.
[MTHM97] Milner, R., M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML — Revised 1997*. The MIT Press, Cambridge, MA, 1997.
[PT93] Pierce, B. C. and D. N. Turner. Statically typed friendly functions via partially abstract types. *Technical Report ECS-LFCS-93-256*, University of Edinburgh, LFCS, April 1993. Also available as INRIA-Rocquencourt Rapport de Recherche No. 1899.
[RS98] Riecke, J. G. and C. Stone. Privacy via subsumption. In *FOOL5*, January 1998. A longer version will appear in TAPOS.
[Str97] Stroustrup, B. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 3rd edition, 1997.

A A brief introduction to MOBY

This appendix provides a brief introduction to some of MOBY's features to help the reader understand the examples in the paper. A more detailed description of MOBY object-oriented features can be found in [FR99a].

MOBY programs are organized into a collection of *modules*, which have *signatures*. A module's signature controls the visibility of its components. Signatures are the primary mechanism for data and type abstraction in MOBY. To support object-oriented programming, MOBY provides *classes* and *object types*. The following example illustrates these features:

```
module M : {
  class Hi : {
    public meth hello : Unit -> Unit
    public maker mk of String
  }
  val hi : typeof(Hi)
} {
  fun pr (s : String) -> Unit { ConsoleIO.print s }
  class Hi {
    field msg : String
    public meth hello () -> Unit {
      pr "hello "; pr self.msg; pr "\n"
    }
    public maker mk (s : String) { field msg = s }
  }
  val hi : typeof(Hi) = new mk "world"
}
```

This code defines a module `M` that is constrained by a signature with two specifications: the class `Hi` and the value `hi`. The interface of the `Hi` class specifies that it has two public components: a method `hello` and a maker `mk` ("maker" is the MOBY name for constructor functions). The signature specifies that `hi` is an object; the type expression "**`typeof`**`(Hi)`" denotes the object type induced by reading off the public methods and fields of the class `Hi`. It is equivalent to the object type definition

```
objtype HiTy { public meth hello : Unit -> Unit }
```

The body of `M` defines a function `pr` for printing strings, and the definitions of `Hi` and `hi`. Since `pr` is not mentioned in the signature of `M`, it is not exported. Note that the `Hi` class has a field `msg` in its definition. Since this field does not have a **`public`** annotation, it is only visible to subclasses. Furthermore, since `msg` is not mentioned in `M`'s signature, it is not visible to subclasses of `Hi` outside of `M`. Thus, the `msg` field is *protected* inside `M` and is *private* outside. This example illustrates MOBY's use of module signatures to implement private class members.

B Dynamic Semantics of XMOC

B.1 Notation

If A and B are two sets, we write $A \setminus B$ for the set difference and $A \pitchfork B$ if they are *disjoint*. We use the notation $A \stackrel{\text{fin}}{\to} B$ to denote the set of finite maps from A to B. We write $\{a_1 \mapsto b_1, \ldots, a_n \mapsto b_n\}$ for the finite map that maps a_1 to b_1, etc. For a finite map f, we write $\text{dom}(f)$ for its domain and $\text{rng}(f)$ for its range. If f and g are finite maps, we write $f \pm g$ for the finite map

$$\{x \mapsto g(x) \mid x \in \text{dom}(g)\} \cup \{x \mapsto f(x) \mid x \in \text{dom}(f) \setminus \text{dom}(g)\}$$

We write $t[x \mapsto t']$ for the *capture-free* substitution of t' for x in the term t.

B.2 Syntax

We use the following classes of identifiers in the syntax of XMOC.

$\alpha \in \textsc{TyVar}$	type variables
$\tau \in \textsc{Type}$	types
$\sigma \in \textsc{Interface}$	class interfaces
$B, C \in \textsc{ClassName} = \{\mathbf{None}, \ldots\}$	class names
$\#B, \#C \in \textsc{ClassType}$	class types
$m \in \textsc{MethName}$	method names
$\mathcal{M} \in \text{Fin}(\textsc{MethName})$	method name sets
$x \in \textsc{Var}$	variables
$\mu \in \textsc{MethBody}$	method bodies
$e \in \textsc{Exp}$	expressions
$obj \in \textsc{ObjExp}$	object expressions

We follow the convention of using C when referring to a class name other than $\mathbf{None}$. Figure 10 describes the full syntax of XMOC. In this grammar, we mark the run-time forms with a $(*)$ on the right. The obj form represents a raw object at run-time: C is its instantiating class, e denotes its local state, and obj represents the portion of the object inherited from its parent class. The methods associated with class C are found in the dynamic class environment, defined below.

$$
\begin{array}{lll}
p & ::= & d;\ p \\
 & \mid & e \\
d & ::= & \texttt{class}\ C\ (x:\tau)\ \{\!|\ b\ \ \overline{m = \mu_m}^{m \in \mathcal{M}}\ |\!\} \\
 & \mid & \texttt{class}\ C : \sigma = C' \\
\sigma & ::= & (\tau)\ \{\!|\ \texttt{inherits}\ B;\ \overline{m : \tau_m}^{m \in \mathcal{M}}\ |\!\} \\
\tau & ::= & \alpha \\
 & \mid & \tau \to \tau' \\
 & \mid & \texttt{obj}\ \alpha\,\texttt{.}\{\!|\ \overline{m : \tau_m}^{m \in \mathcal{M}}\ |\!\} \\
 & \mid & \texttt{\#}B \\
b & ::= & \\
 & \mid & \texttt{inherits}\ C(e);
\end{array}
\qquad
\begin{array}{llll}
\mu & ::= & (x\ :\ \tau) \Rightarrow e & \\
e & ::= & x & \\
 & \mid & \texttt{fn}(x\ :\ \tau) \Rightarrow e & \\
 & \mid & e(e') & \\
 & \mid & \texttt{new}\ C(e) & \\
 & \mid & \texttt{self} & \\
 & \mid & e\,\texttt{!state} & \\
 & \mid & e\,@\,C & \\
 & \mid & e\,\texttt{.}\,m & \\
 & \mid & \langle obj,\ C\rangle & (*) \\
obj & ::= & \textbf{None} & (*) \\
 & & C :: \{\!|\ e;\ obj\ |\!\} & (*) \\
 & & C(e) & (*)
\end{array}
$$

Fig. 10. The full syntax of XMOC

B.3 Evaluation

To define the evaluation relation, we need some additional definitions:

$$
\begin{array}{rll}
 & \textsc{Slot} & \text{method suite slots} \\
\phi \in & \textsc{Dict} = \textsc{MethName} \xrightarrow{\text{fin}} \textsc{Slot} & \text{dictionaries} \\
\mu T \in & \textsc{MethSuite} = \textsc{Slot} \xrightarrow{\text{fin}} \textsc{MethBody} & \text{method suites} \\
H \in & \textsc{Hierarchy} = \textsc{ClassName} \xrightarrow{\text{fin}} \textsc{ClassName} & \text{class hierarchy}
\end{array}
$$

A *dictionary* ϕ is a one-to-one finite function from method names to slots. We say that ϕ is a *slot assignment* for a set of method names M if and only if $\mathrm{dom}(\phi) = M$. A *method suite* is a finite function from slots to method bodies. A class hierarchy describes the inheritance relationship between classes by mapping each class name to the name of its superclass.

The dynamic semantics is split into two phases; the first phase *links* the class declarations to produce a dynamic class environment and a residual expression. The dynamic class environment is a finite map from class names to a tuple, storing the name of the parent class, a constructor expression for initializing the inherited state of instantiated objects, a method suite, which maps slot numbers to method bodies, and a dictionary, which maps method names to slot numbers.

$$
\mathcal{K} \in \textsc{DynClassEnv} = \textsc{ClassName} \xrightarrow{\text{fin}} (\textsc{ClassName} \times \textsc{Exp} \times \textsc{MethSuite} \times \textsc{Dict})
$$

We write the linking relation as $\mathcal{K}, p \rightarrowtail \mathcal{K}', p'$. The second phase *evaluates* the residual expression using the dynamic class environment to instantiate objects and resolve method dispatch. The evaluation relation is written formally as $\mathcal{K}, e \rightarrowtail \mathcal{K}, e'$, but we abbreviate this notation to $\mathcal{K} \vdash e \hookrightarrow e'$ when convenient.

B.4 Dynamic Class Hierarchy

We construct the dynamic class hierarchy from the dynamic class environment $\mathcal{K}$ as follows:

$$\mathcal{H}(\mathcal{K}) = \{C \mapsto B \mid C \in \text{dom}(\mathcal{K}) \text{ and } \mathcal{K}(C) = (B,\ \_,\ \_,\ \_)\}$$

B.5 Evaluation Contexts and Values

The dynamic semantics is specified using the standard technique of evaluation contexts. We distinguish two kinds of contexts in the specification of the evaluation relation: *expression contexts*, E, and *object initialization contexts*, F. The syntax of these contexts is as follows:

$$\begin{aligned} E &::= [\,] \mid E(e) \mid w(E) \mid \mathtt{\textbf{new}}\ C(E) \mid E\,\mathtt{\textbf{!state}} \mid E\,@\,C \mid E\,\mathtt{.}\,m \mid \langle F,\ C\rangle \\ F &::= [\,] \mid C(E) \mid C :: \{\!|\, E;\ obj\,|\!\} \mid C :: \{\!|\, w;\ F\,|\!\} \end{aligned}$$

We also define the syntactic class of *values* (w) and *object values* (ov) by the following grammar:

$$\begin{aligned} w &::= \mathtt{\textbf{fn}}(x\ :\ \tau) \Rightarrow e \mid \langle ov,\ C\rangle \\ ov &::= \mathbf{None} \mid C :: \{\!|\, w;\ ov\,|\!\} \end{aligned}$$

B.6 Field lookup

The auxiliary *field lookup* relation $C_v \vdash C :: \{\!|\, w;\ ov\,|\!\} \leadsto_f w'$ is used to specify the semantics of state selection.

$$\frac{}{C \vdash C :: \{\!|\, w;\ ov\,|\!\} \leadsto_f w} \quad (1) \qquad \frac{C_v \neq C \qquad C_v \vdash ov \leadsto_f w'}{C_v \vdash C :: \{\!|\, w;\ ov\,|\!\} \leadsto_f w'} \quad (2)$$

B.7 Method lookup

We define the auxiliary *method lookup* relation $\mathcal{K} \vdash \langle ov,\ C_v\rangle\,\mathtt{.}\,m \leadsto_m w$, which specifies how method dispatching is resolved, as follows:

$$\frac{\mathcal{K}(C_v) = (\_,\ \_,\ \_,\ \phi) \qquad \mathcal{K}(C) = (\_,\ \_,\ \mu T,\ \_) \qquad \mu T(\phi(m)) = (x\ :\ \tau) \Rightarrow e}{\mathcal{K} \vdash \langle C :: \{\!|\, w;\ ov\,|\!\},\ C_v\rangle\,\mathtt{.}\,m \leadsto_m \mathtt{\textbf{fn}}(x\ :\ \tau) \Rightarrow e[\mathit{self} \mapsto C :: \{\!|\, w;\ ov\,|\!\}]} \quad (3)$$

B.8 Class linking

The evaluation relation for class linking is defined by the following three rules. The first describes the case of a base-class declaration. Note that since a base class has no superclass, it does not require a parent initialization expression.

$$\begin{array}{l} \mathcal{K}, \mathtt{class}\ C\ (x : \tau)\,\{\!\!|\ m = \mu_m{}^{m\in\mathcal{M}}\ |\!\!\};\ p\ \rightarrowtail\ \mathcal{K}\pm\{C \mapsto (\mathbf{None},\ -,\ \mu T,\ \phi)\}, p \\ \text{where}\ \overline{\mu}_m = \mu_m[\mathit{self} \mapsto \mathit{self} \mathbin{@} C] \\ \qquad \phi \text{ is a slot assignment for } M. \\ \qquad \mu T = \{\phi(m) \mapsto \overline{\mu}_m | m \in M\} \end{array} \tag{4}$$

The second rule handles the case of a subclass declaration.

$$\begin{array}{l} \mathcal{K}, \mathtt{class}\ C\ (x : \tau)\,\{\!\!|\,\mathtt{inherits}\,C'(e);\ m = \mu_m{}^{m\in\mathcal{M}}\ |\!\!\};\ p \\ \quad \rightarrowtail \quad \mathcal{K}\pm\{C \mapsto (C',\ \mathtt{fn}(x\ :\ \tau) \Rightarrow e,\ \mu T_C,\ \phi_C)\}, p \\ \text{where}\ \mathcal{K}(C') = (\_,\ \_,\ \mu T_{C'},\ \phi_{C'}) \\ \qquad \phi \text{ is a slot assignment for } M \setminus \text{dom}(\phi_{C'}) \text{ such that } \text{rng}(\phi) \pitchfork \text{dom}(\mu T_{C'}) \\ \qquad \phi_C = \phi_{C'} \cup \phi \\ \qquad \overline{\mu}_m = \mu_m[\mathit{self} \mapsto \mathit{self} \mathbin{@} C] \\ \qquad \mu T_C = \mu T_{C'} \pm \{\phi_C(m) \mapsto \overline{\mu}_m | m \in M\} \end{array} \tag{5}$$

The third linking rule describes class signature ascription.

$$\begin{array}{l} \mathcal{K}, \mathtt{class}\ C : (\tau)\,\{\!\!|\,\mathtt{inherits}\,B;\ m : \tau_m{}^{m\in\mathcal{M}}\ |\!\!\} = C';\ p \\ \quad \rightarrowtail \quad \mathcal{K}\pm\{C \mapsto (C',\ \mathtt{fn}(x\ :\ \tau) \Rightarrow x,\ \mu T_C,\ \phi_C)\}, p \\ \text{where}\ \mathcal{K}(C') = (\_,\ \_,\ \mu T_{C'},\ \phi_{C'}) \\ \qquad \mu T_C = \mu T_{C'} \\ \qquad \phi_C = \phi_{C'} | M \end{array} \tag{6}$$

B.9 Expression evaluation

The following rules specify the semantics of expression evaluation:

$$\mathcal{K} \vdash E[\mathtt{fn}(x\ :\ \tau) \Rightarrow e(w)] \quad \hookrightarrow \quad E[e[x \mapsto w]] \tag{7}$$

$$\mathcal{K} \vdash E[\mathtt{new}\ C(w)] \quad \hookrightarrow \quad E[\langle C(w),\ C\rangle] \tag{8}$$

$$\mathcal{K} \vdash E[\langle ov,\ C_v\rangle\,\mathtt{!state}] \quad \hookrightarrow \quad E[w] \qquad \text{where } C_v \vdash ov \rightsquigarrow_f w \tag{9}$$

$$\mathcal{K} \vdash E[\langle ov,\ C_v\rangle \mathbin{@} C'_v] \quad \hookrightarrow \quad E[\langle ov,\ C'_v\rangle] \tag{10}$$

$$\mathcal{K} \vdash E[w\,\mathbf{.}\,m] \quad \hookrightarrow \quad E[w'] \qquad \text{where } \mathcal{K} \vdash w\,\mathbf{.}\,m \rightsquigarrow_m w' \tag{11}$$

$$\begin{array}{l} \mathcal{K} \vdash F[C(w)] \quad \hookrightarrow \quad F[C :: \{\!\!|\,w;\ \mathit{obj}\,|\!\!\}] \\ \text{where} \quad \mathit{obj} = \begin{cases} \mathbf{None} & \text{if } \mathcal{K}(C) = (\mathbf{None},\ \_,\ \_,\ \_) \\ C'(e(w)) & \text{if } \mathcal{K}(C) = (C',\ e,\ \_,\ \_) \end{cases} \end{array} \tag{12}$$

C Typing rules for XMOC

The typing rules for XMOC are written with respect to an environment Γ, which has four parts:

$\mathcal{A} \in \textsc{TyVarSet} = \mathrm{Fin}(\textsc{TyVar})$	bound type variables
$\mathcal{S} \in \textsc{SubtyEnv} = \textsc{TyVar} \overset{\text{fin}}{\to} \textsc{Type}$	subtyping assumptions
$\mathcal{C} \in \textsc{ClassEnv} = \textsc{ClassName} \overset{\text{fin}}{\to} \textsc{Interface}$	class typing environment
$\mathcal{V} \in \textsc{VarEnv} = \textsc{Var} \overset{\text{fin}}{\to} \textsc{Type}$	variable typing environment
$\Gamma \in \textsc{Env} = \textsc{TyVarSet} \times \textsc{SubtyEnv} \times \textsc{ClassEnv} \times \textsc{VarEnv}$	typing environment

C.1 Static Class Hierarchy

We construct the static class hierarchy from the environment Γ as follows:

$$\mathcal{H}(\Gamma) = \{(C \mapsto B) \mid (\mathcal{C}\,\mathrm{of}\,\Gamma)(C) = (\tau)\,\{\!|\,\mathtt{inherits}\,B;\ m : {\tau_m}^{m \in \mathcal{M}}\,|\!\}\}$$

C.2 Judgement forms

$\Gamma \vdash \tau \triangleright \mathbf{Ok}$	Type τ is *well-formed* w.r.t. Γ.	(rules 13–17)
$\Gamma \vdash \sigma \triangleright \mathbf{Ok}$	Class interface σ is *well-formed* w.r.t. Γ.	(rule 18)
$H \vdash \mathbf{Ok}$	Class hierarchy H is *well-formed.*	(rule 19)
$\Gamma \vdash \mathbf{Ok}$	Environment Γ is *well-formed.*	(rule 20)
$\Gamma \vdash \tau = \tau'$	Type τ is *equal* to τ'.	(rules 21–26)
$H \vdash B_1 \lessdot B_2$	Class B_1 *inherits* from B_2.	(rules 27–30)
$\Gamma \vdash \tau <: \tau'$	Type τ is a *subtype* of τ'.	(rules 31–36)
$\Gamma \vdash \sigma <: \sigma'$	Class interface σ is a *subtype* of σ'.	(rule 37)
$\Gamma \vdash p \triangleright \Gamma', \tau$	Program p *defines* environment Γ' and *has type* τ.	(rules 38–39)
$\Gamma \vdash d \triangleright \Gamma'$	Declaration d *defines* environment Γ'.	(rules 40–42)
$\Gamma \vdash \mu \triangleright \tau$	Method μ *has type* τ.	(rule 43)
$\Gamma \vdash e \triangleright \tau$	Expression e *has type* τ.	(rules 44–51)
$\Gamma, H \vdash obj \triangleright \#B$	Parent class obj *has class type* $\#B$.	(rules 52–54)
$\Gamma, H \vdash e \triangleright \tau$	Run-time expression e *has type* τ.	(rules 53 and 56)
$\Gamma, H, C \vdash con \triangleright \sigma$	Constructor con, associated with class C, *has interface* σ.	(rules 57–58)
$\Gamma \vdash \mathcal{K}$	Static environment Γ *types* dynamic class environment $\mathcal{K}$.	(rule 59)

C.3 Well-formedness judgements

$$\frac{\Gamma \vdash \mathbf{Ok} \quad \alpha \in (\mathcal{A}\,\mathrm{of}\,\Gamma)}{\Gamma \vdash \alpha \triangleright \mathbf{Ok}} \qquad (13)$$

$$\frac{\Gamma \vdash \mathbf{Ok}}{\Gamma \vdash \#\mathbf{None} \triangleright \mathbf{Ok}} \qquad (14)$$

$$\frac{\Gamma \vdash \tau \triangleright \mathbf{Ok} \quad \Gamma \vdash \tau' \triangleright \mathbf{Ok}}{\Gamma \vdash \tau \to \tau' \triangleright \mathbf{Ok}} \qquad (15)$$

$$\frac{\Gamma \vdash \mathbf{Ok} \quad \#C \in \mathrm{dom}(\mathcal{C}\,\mathrm{of}\,\Gamma)}{\Gamma \vdash \#C \triangleright \mathbf{Ok}} \qquad (16)$$

$$\frac{\Gamma \vdash \mathbf{Ok} \quad \forall m \in \mathcal{M}\ \Gamma \cup \{\alpha\} \vdash \tau_m \triangleright \mathbf{Ok}}{\Gamma \vdash \texttt{obj}\, \alpha\,\texttt{.} \{\!|\, m : \tau_m{}^{m \in \mathcal{M}} \,|\!\} \triangleright \mathbf{Ok}} \tag{17}$$

$$\frac{\Gamma \vdash \tau \triangleright \mathbf{Ok} \quad \Gamma \vdash \texttt{\#}B \triangleright \mathbf{Ok} \quad \forall m \in \mathcal{M}\ \Gamma \vdash \tau_m \triangleright \mathbf{Ok}}{\Gamma \vdash (\tau)\, \{\!|\, \texttt{inherits}\, B;\ m : \tau_m{}^{m \in \mathcal{M}} \,|\!\} \triangleright \mathbf{Ok}} \tag{18}$$

$$\frac{\forall C \in \mathrm{dom}(H) \quad \mathcal{H}(C) \in \mathrm{dom}(H) \cup \{\mathbf{None}\}}{H \vdash \mathbf{Ok}} \tag{19}$$

$$\frac{\begin{array}{c}\forall \tau \in \mathrm{rng}(\mathcal{S}\ \mathrm{of}\ \Gamma)\ \Gamma \vdash \tau \triangleright \mathbf{Ok} \\ \forall \sigma \in \mathrm{rng}(\mathcal{C}\ \mathrm{of}\ \Gamma)\ \Gamma \vdash \sigma \triangleright \mathbf{Ok} \\ \forall \tau' \in \mathrm{rng}(\mathcal{V}\ \mathrm{of}\ \Gamma)\ \Gamma \vdash \tau' \triangleright \mathbf{Ok}\end{array}}{\Gamma \vdash \mathbf{Ok}} \tag{20}$$

C.4 Equality judgements

$$\frac{\Gamma \vdash \tau \triangleright \mathbf{Ok}}{\Gamma \vdash \tau = \tau} \tag{21}$$

$$\frac{\Gamma \vdash \tau' = \tau}{\Gamma \vdash \tau = \tau'} \tag{22}$$

$$\frac{\Gamma \vdash \tau = \tau' \quad \Gamma \vdash \tau' = \tau''}{\Gamma \vdash \tau = \tau''} \tag{23}$$

$$\frac{\Gamma \vdash \tau_1 = \tau_2 \quad \Gamma \vdash \tau_1' = \tau_2'}{\Gamma \vdash \tau_1 \to \tau_1' = \tau_2 \to \tau_2'} \tag{24}$$

$$\frac{\mathcal{M}' = \mathcal{M} \quad \forall m \in \mathcal{M}\ \Gamma \cup \{\alpha\} \vdash \tau_m = \tau_m'}{\Gamma \vdash \texttt{obj}\, \alpha\,\texttt{.} \{\!|\, m : \tau_m{}^{m \in \mathcal{M}} \,|\!\} = \texttt{obj}\, \alpha\,\texttt{.} \{\!|\, m : \tau'{}_m{}^{m \in \mathcal{M}'} \,|\!\}} \tag{25}$$

$$\frac{\Gamma \vdash \texttt{obj}\, \alpha\,\texttt{.} \{\!|\, m : \tau_m{}^{m \in \mathcal{M}} \,|\!\} \triangleright \mathbf{Ok}}{\begin{array}{l}\Gamma \vdash \texttt{obj}\, \alpha\,\texttt{.} \{\!|\, m : \tau_m{}^{m \in \mathcal{M}} \,|\!\} \\ \qquad = \texttt{obj}\, \alpha\,\texttt{.} \{\!|\, m : \tau_m[\alpha \mapsto \texttt{obj}\, \alpha\,\texttt{.} \{\!|\, m : \tau_m{}^{m \in \mathcal{M}} \,|\!\}]^{m \in \mathcal{M}} \,|\!\}\end{array}} \tag{26}$$

C.5 Class hierarchy judgements

$$\frac{B \in \mathrm{dom}(H) \cup \{\mathbf{None}\}}{H \vdash B \lessdot B} \tag{27}$$

$$\frac{C \in \mathrm{dom}(H)}{H \vdash C \lessdot \mathbf{None}} \tag{28}$$

$$\frac{H \vdash B_1 \lessdot B_2 \qquad H \vdash B_2 \lessdot B_3}{H \vdash B_1 \lessdot B_3} \tag{29}$$

$$\frac{H(C) = B}{H \vdash C \lessdot B} \tag{30}$$

C.6 Subtyping judgements

$$\frac{\Gamma \vdash \tau = \tau'}{\Gamma \vdash \tau <: \tau'} \quad (31) \qquad \frac{\Gamma \vdash \tau <: \tau' \quad \Gamma \vdash \tau' <: \tau''}{\Gamma \vdash \tau <: \tau''} \quad (32)$$

$$\frac{\Gamma \vdash \mathbf{Ok} \quad \alpha \in \mathrm{dom}(\mathcal{S} \text{ of } \Gamma)}{\Gamma \vdash \alpha <: \mathcal{S}(\alpha)} \quad (33) \qquad \frac{\Gamma \vdash \tau_2 <: \tau_1 \quad \Gamma \vdash \tau_1' <: \tau_2'}{\Gamma \vdash \tau_1 \to \tau_1' <: \tau_2 \to \tau_2'} \quad (34)$$

$$\frac{\begin{array}{c}\mathcal{M}' \subseteq \mathcal{M} \quad \alpha' \notin \mathrm{FV}(\mathtt{obj}\,\alpha\,\mathtt{.}\,\{\!|\, m : \tau_m{}^{m \in \mathcal{M}} \,|\!\})\\ \forall m \in \mathcal{M}'\ \alpha \notin \mathrm{FV}(\tau_m') \text{ and } (\Gamma \cup \{\alpha'\}) \pm \{\alpha \mapsto \alpha'\} \vdash \tau_m <: \tau_m'\end{array}}{\Gamma \vdash \mathtt{obj}\,\alpha\,\mathtt{.}\,\{\!|\, m : \tau_m{}^{m \in \mathcal{M}} \,|\!\} <: \mathtt{obj}\,\alpha'\,\mathtt{.}\,\{\!|\, m : \tau'_m{}^{m \in \mathcal{M}'} \,|\!\}} \quad (35)$$

$$\frac{\Gamma \vdash \mathbf{Ok} \quad (\mathcal{C} \text{ of } \Gamma)(C) = (\tau)\,\{\!|\, \mathtt{inherits}\, B;\ m : \tau_m{}^{m \in \mathcal{M}} \,|\!\}}{\Gamma \vdash \mathtt{\#}C <: \mathtt{obj}\,\alpha\,\mathtt{.}\,\{\!|\, m : \tau_m{}^{m \in \mathcal{M}} \,|\!\}} \quad (36)$$

$$\frac{\begin{array}{c}\Gamma \vdash \tau' <: \tau \quad H \vdash B \lessdot B' \quad \mathcal{M}' \subseteq \mathcal{M} \quad \forall m \in \mathcal{M}'\ \Gamma \vdash \tau_m = \tau_m'\\ \forall m \in (\mathcal{M} \setminus \mathcal{M}')\ \Gamma \vdash \tau_m \rhd \mathbf{Ok}\end{array}}{\Gamma \vdash (\tau)\,\{\!|\, \mathtt{inherits}\, B;\ m : \tau_m{}^{m \in \mathcal{M}} \,|\!\} <: (\tau')\,\{\!|\, \mathtt{inherits}\, B';\ m : \tau'_m{}^{m \in \mathcal{M}'} \,|\!\}} \quad (37)$$

C.7 Typing judgements

As a notational convenience, we define the argument type of a class C in an environment Γ (written $\mathrm{ArgTy}(\Gamma, C)$) to be τ if $\Gamma(C) = (\tau)\,\{\!|\, \mathtt{inherits}\, B;\ m : \tau_m{}^{m \in \mathcal{M}} \,|\!\}$.

$$\frac{\Gamma \vdash d \rhd \Gamma' \quad \Gamma' \vdash p \rhd \Gamma'', \tau}{\Gamma \vdash d;\ p \rhd \Gamma'', \tau} \quad (38) \qquad \frac{\Gamma \vdash e \rhd \tau}{\Gamma \vdash e \rhd \Gamma, \tau} \quad (39)$$

$$\frac{\begin{array}{c}C \notin \mathrm{dom}(\mathcal{C} \text{ of } \Gamma) \quad \Gamma \vdash \tau \rhd \mathbf{Ok}\\ \Gamma' = \Gamma \pm \{C \mapsto (\tau)\,\{\!|\, \mathtt{inherits}\, \mathrm{None};\ m : \tau_m{}^{m \in \mathcal{M}} \,|\!\}\}\\ \forall m \in \mathcal{M}\ \Gamma' \pm \{\mathit{self} \mapsto \mathtt{\#}C\} \vdash \mu_m \rhd \tau_m' \qquad \Gamma' \vdash \tau_m' <: \tau_m\end{array}}{\Gamma \vdash \mathtt{class}\ C\ (x : \tau)\,\{\!|\ m = \mu_m{}^{m \in \mathcal{M}} \,|\!\} \rhd \Gamma'} \quad (40)$$

$$\frac{\begin{array}{c}C \notin \mathrm{dom}(\mathcal{C} \text{ of } \Gamma)\\ (\mathcal{C} \text{ of } \Gamma)(C') = (\tau')\,\{\!|\, \mathtt{inherits}\, B';\ m : \tau'_m{}^{m \in \mathcal{M}'} \,|\!\}\\ \Gamma' = \Gamma \pm \{C \mapsto (\tau)\,\{\!|\, \mathtt{inherits}\, C';\ m : \tau'_m{}^{m \in (\mathcal{M}' \setminus \mathcal{M})}\ m : \tau_m{}^{m \in \mathcal{M}} \,|\!\}\}\\ \Gamma' \pm \{x \mapsto \tau\} \vdash e \rhd \tau'' \quad \Gamma' \vdash \tau'' <: \tau'\\ \forall m \in (\mathcal{M} \cap \mathcal{M}')\ \Gamma' \vdash \tau_m <: \tau_m'\\ \forall m \in \mathcal{M}\ \Gamma' \pm \{\mathit{self} \mapsto \mathtt{\#}C\} \vdash \mu_m \rhd \tau_m'' \qquad \Gamma' \vdash \tau_m'' <: \tau_m\end{array}}{\Gamma \vdash \mathtt{class}\ C\ (x : \tau)\,\{\!|\, \mathtt{inherits}\, C'(e);\ m = \mu_m{}^{m \in \mathcal{M}} \,|\!\} \rhd \Gamma'} \quad (41)$$

$$\frac{C \notin \mathrm{dom}(\mathcal{C}\text{ of }\Gamma) \quad \Gamma' = \Gamma \pm \{C \mapsto \sigma\} \quad \Gamma' \vdash (\mathcal{C}\text{ of }\Gamma)(C') <: \sigma}{\Gamma \vdash \mathtt{class}\ C : \sigma = C' \triangleright \Gamma'} \tag{42}$$

$$\frac{\Gamma \pm \{x \mapsto \tau\} \vdash e \triangleright \tau'}{\Gamma \vdash (x\ :\ \tau) \Rightarrow e \triangleright \tau \to \tau'} \tag{43}$$

$$\frac{\Gamma \vdash \mathbf{Ok} \quad (\mathcal{V}\text{ of }\Gamma)(x) = \tau}{\Gamma \vdash x \triangleright \tau} \tag{44}$$

$$\frac{\Gamma \vdash \mathbf{Ok} \quad (\mathcal{V}\text{ of }\Gamma)(\mathtt{self}) = \tau}{\Gamma \vdash \mathtt{self} \triangleright \tau} \tag{45}$$

$$\frac{\Gamma \pm \{x \mapsto \tau\} \vdash e \triangleright \tau'}{\Gamma \vdash \mathtt{fn}(x\ :\ \tau) \Rightarrow e \triangleright \tau \to \tau'} \tag{46}$$

$$\frac{\Gamma \vdash e \triangleright \tau' \to \tau \quad \Gamma \vdash e' \triangleright \tau'' \quad \Gamma \vdash \tau'' <: \tau'}{\Gamma \vdash e(e') \triangleright \tau} \tag{47}$$

$$\frac{\Gamma \vdash e \triangleright \tau' \quad \Gamma \vdash \tau' <: \mathrm{ArgTy}(\Gamma,\, C)}{\Gamma \vdash \mathtt{new}\ C(e) \triangleright \#C} \tag{48}$$

$$\frac{\Gamma \vdash e \triangleright \#C}{\Gamma \vdash e\mathtt{!state} \triangleright \mathrm{ArgTy}(\Gamma,\, C)} \tag{49}$$

$$\frac{\Gamma \vdash e \triangleright \#C' \quad \mathcal{H}(\Gamma) \vdash C' \lessdot C}{\Gamma \vdash e\,@\,C \triangleright \#C} \tag{50}$$

$$\frac{\Gamma \vdash e \triangleright \tau \quad m \in \mathcal{M} \quad \Gamma \vdash \tau <: \mathtt{obj}\,\alpha\,\mathtt{.}\,\{\!|\, m : \tau_m{}^{m \in \mathcal{M}} \,|\!\}}{\Gamma \vdash e\,\mathtt{.}\,m \triangleright \tau_m[\alpha \mapsto \mathtt{obj}\,\alpha\,\mathtt{.}\,\{\!|\, m : \tau_m{}^{m \in \mathcal{M}} \,|\!\}]} \tag{51}$$

C.8 Typing rules for run-time forms

The typing rules for the run-time forms include rules 43 through 51 augmented with a dynamic class hierarchy H on the left-hand side of the turnstile. Rules 40 through 42 return an augmented class hierarchy in addition to an augmented static environment. In each case, the augmented hierarchy is just the old one extended with a mapping from the newly declared class name to the name of its parent (or **None** for base classes). In addition, rule 50 is replaced by rule 55. The remaining rules are given below.

$$\frac{\Gamma, H \vdash e \triangleright \tau' \quad \Gamma \vdash \tau' <: \mathrm{ArgTy}(\Gamma,\, C)}{\Gamma, H \vdash C(e) \triangleright \#C} \tag{52}$$

$$\frac{H(C) = B \quad \Gamma, H \vdash obj \triangleright \#B \quad \Gamma, H \vdash e \triangleright \tau' \quad \Gamma \vdash \tau' <: \mathrm{ArgTy}(\Gamma,\, C)}{\Gamma, H \vdash C :: \{\!|\, e;\ obj \,|\!\} \triangleright \#C} \tag{53}$$

$$\frac{\Gamma \vdash \mathbf{Ok} \quad \mathcal{K} \vdash \mathbf{Ok}}{\Gamma, H \vdash \mathbf{None} \triangleright \#\mathbf{None}} \tag{54}$$

$$\frac{\Gamma, H \vdash e \triangleright \#C' \quad H \vdash C' \lessdot C}{\Gamma, H \vdash e\,@\,C \triangleright \#C} \quad (55) \qquad \frac{\Gamma, H \vdash obj \triangleright \#C' \quad H \vdash C' \lessdot C}{\Gamma, H \vdash \langle obj,\ C\rangle \triangleright \#C} \quad (56)$$

$$\frac{\begin{array}{c}\sigma = (\tau)\,\{\!\!\{\, \texttt{\textbf{inherits}}\, \mathbf{None};\ m : \tau_m{}^{m \in \mathcal{M}}\,\}\!\!\}\\ \mathcal{M} = \mathrm{dom}(\phi) \qquad \mathrm{rng}(\phi) \subseteq \mathrm{dom}(\mu T)\\ \forall m \in\ \mathcal{M}\ \Gamma \pm \{self \mapsto \#C\}, H \vdash \mu T(\phi(m)) \triangleright \tau'_m\\ \Gamma \vdash \tau'_m <: \tau_m\end{array}}{\Gamma, H, C \vdash (\mathbf{None},\ ??,\ \mu T,\ \phi) \triangleright \sigma} \quad (57)$$

$$\frac{\begin{array}{c}\sigma = (\tau)\,\{\!\!\{\, \texttt{\textbf{inherits}}\, C';\ m : \tau_m{}^{m \in \mathcal{M}}\,\}\!\!\}\\ \Gamma(C') = (\tau')\,\{\!\!\{\, \texttt{\textbf{inherits}}\, B';\ m : \tau'{}_m{}^{m \in \mathcal{M}'}\,\}\!\!\}\\ \Gamma, H \vdash e \triangleright \tau \to \tau'' \quad \Gamma \vdash \tau'' <: \tau'\\ \mathcal{M} = \mathrm{dom}(\phi) \qquad \mathrm{rng}(\phi) \subseteq \mathrm{dom}(\mu T)\\ \forall m \in\ \mathcal{M}\ \Gamma \pm \{self \mapsto \#C\}, H \vdash \mu T(\phi(m)) \triangleright \tau''_m\\ \Gamma \vdash \tau''_m <: \tau_m\\ \forall m \in\ \mathcal{M} \cap \mathcal{M}'\ \Gamma \vdash \tau_m <: \tau'_m\end{array}}{\Gamma, H, C \vdash (C',\ e,\ \mu T,\ \phi) \triangleright \sigma} \quad (58)$$

$$\frac{\begin{array}{c}\mathrm{dom}(\mathcal{K}) = \mathrm{dom}(\mathcal{C} \text{ of } \Gamma)\\ \forall C \in \mathrm{dom}(\mathcal{K})\ \ \Gamma(C) = \sigma \text{ and } \Gamma, H(\mathcal{K}), C \vdash \mathcal{K}(C) \triangleright \sigma' \text{ and } \Gamma \vdash \sigma' <: \sigma\end{array}}{\Gamma \vdash \mathcal{K}} \quad (59)$$